DEPARTMENT OF PHYSICS
UNIVERSITY OF CAPE TOWN
IYUNIVESITHI YASEKAPA • UNIVERSITEIT VAN KAAPSTAD

Parallel computing, benchmarking and ATLAS software on ARM

University of Cape Town

Joshua Wyatt Smith

Supervised by: Dr. Andrew Hamilton

A dissertation submitted to the University of Cape Town for the degree of Master of Science in Physics

June 2015

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

Abstract

This thesis explores the use of the ARM architecture in high energy physics computing. ARM processors are predominantly found in smartphones and mobile tablets. Results from benchmarks which were performed on the armv7l architecture are presented. These provide qualitative data as well as confirmation that specialized high energy physics software does run on ARM. This thesis presents the first ever port of the ATLAS software stack to the ARM architecture, as well as the issues that ensued. A new framework, ANA, is introduced which facilitates the compilation of the ATLAS software stack on ARM.

Declaration

I know the meaning of plagiarism and declare that all of the work in the dissertation, save for that which is properly acknowledged, is my own.

Joshua Wyatt Smith, 12 June 2015

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Andrew Hamilton. Thank you for introducing me to the Raspberry Pi, from that moment on I've been hooked on this research. He has provided me with opportunities that I would not have thought possible. For that I am, and always will be grateful.

In the beginning, I thought "log" files were provided with the sole purpose of confusing developers. Thank you, Dr. Tom Dietel for showing me the light and for teaching me the ins and outs of Linux. Also, I'm sorry I broke the server.

I need to thank Clint Sadler and Kerwin Ontong for solving all my electrical problems. Thank you, Clint, for building attachments and providing advice so that I didn't electrocute myself. I'm also very appreciative to Clint and Kerwin for organizing a battery back-up to keep my development boards running in the darkest of times (i.e. through load-shedding in Cape Town).

Dr. Rolf Seuster at CERN is evidently a very patient man. Thank you, for finding the time to provide valuable advice, even to someone who in the beginning didn't know "what an Athena is".

I am also grateful to Dr. Carlos Sanchez who is also at CERN. He provided guidance (and code) for the E/p analysis. Crucial tips came in the form of using CMT which helped in the development of ANA.

Thank you to The National Research Foundation of South Africa and the University of Cape Town for funding.

Finally, thank you to my family for all the support and love they have provided over the years. Obviously, none of this would have been possible without them. Also, thank you for proof reading this thesis and providing valuable insight.

Contents

List of Figures

Chapter 1

Introduction

The Large Hadron Collider (LHC) [1] is currently the world's largest scientific experiment. It is a particle collider in which two beams traverse a 27 km long tunnel in opposite directions colliding protons and heavy ions (such as lead) at four interaction points (IP's). Detectors are placed at each IP where they observe these collisions. The hardware and software inside the detectors extract the interesting events and records them for later study. There is an ever increasing need to create and store more data as rarer particles and processes are only detectable at higher beam intensities and energies. An example is the Higgs Boson discovery of 2012 [2, 3] made at the CMS [4] and ATLAS [5] detectors. In order for these discoveries to be made, more than 170 computer farms from 40 countries share resources with the common goal of analyzing the data that the experiments create and store. The Worldwide LHC Grid (LHC Grid) [6] was commissioned to enable such resource sharing to take place and currently has over 160 thousand traditional x86 processors.

Following the completion of Run 1 in early 2013, the LHC has now commenced with Run 2, where beam energies have been increased by almost a factor of two. Run 3 has been approved and will commence around early 2020. Research has to be done to determine how to reach higher energies and thus, how to handle the increased volume of data. The problem of how to handle "big data" is global and multidisciplinary and extends beyond physics. For the past fifty years there has been an exponential increase in the performance of computers as predicted by Moore's Law [7]. It essentially says that computer power doubles every two years due to being able to fit more transistors in a circuit. However, Moore himself amended his initial observation and proceeded to state that the semiconductor industry could not maintain its exponential growth [8]. Furthermore,

performance per watt is now starting to become a higher priority as the cost to run high performance server farms escalates. In the general case of the Internet:

> *"It is estimated that data centers delivering Internet services consume over 1.5 percent of U.S. electric power. As the use of the Internet continues to grow and massive computing facilities are demanding that performance keep doubling, devoting corresponding increases in the nation's electrical energy capacity to computing may become too expensive."* [9]

1.5 percent of the total U.S. electric power translates to approximately 6.5 GW.[1] In order for the next generation of "big data" computing to succeed, a paradigm shift needs to take place so that this enormous scale of computing is feasible. In the search for Standard Model (SM) and Beyond Standard Model (BSM) physics, there is no upper bound on computing power. More servers means that more data can be analyzed in a shorter amount of time. The only major cap is the cost of such server farms. At the scale of computing done at the LHC, running a server farm is a costly affair. Aside from acquiring the hardware, costs in the form of power consumption escalate due to the central processing units (CPU's), peripherals and the cooling systems. The question arises: How we can minimize costs for running a server farm? This will allow more institutions to have them, and thus increase the LHC computing capacity.

One proposed solution is to make use of ARM® processors (from here on referred to as ARM processors). These are found in smartphones and tablet computers where the combination of low power consumption and high performance is the top priority. Significant savings might be achieved if ARM processors will be able to cope with the huge amount of data processed. This thesis serves to explore the capabilities of ARM processors through parallel processing benchmarks as well as compiling and running high energy physics (HEP) software for the ATLAS experiment.

1.1 The Standard Model

According to the Standard Model (SM) of particle physics [10, 11, 12], our universe is composed of twelve fundamental particles (not counting their respective antiparticles) which are called *fermions*. These are spin-1/2 particles that can be further classified into

[1] Statistics were taken from http://www.eia.gov/ for 2012.

two groups: The *quarks* which consist of the up (u), down (d), charm (c), strange (s), top (t) and bottom (b) and the *leptons* which consist of the electron (e), muon (μ), tau (τ), electron neutrino (ν_e), muon neutrino (ν_μ) and tau neutrino (ν_τ). The SM describes four spin-1 gauge bosons called the gluon (g), photon (γ), Z and W, and one spin-0 (or scalar) boson which is the recently discovered Higgs particle. These particles are shown in Figure 1.1.

Quarks carry colour charge, fractional electric charge and weak isospin and therefore interact via the strong interaction (mediated by gluons), the electromagnetic interaction (mediated by photons) and the weak interaction (mediated by W and Z bosons). Due to the constraints that colour charge and confinement provide, quarks cannot be seen as free particles but rather form baryons or mesons which consist of either three colour neutral quarks or a quark-antiquark pair, respectively. Examples of baryons include the Proton (uud) and the Neutron (udd), while examples for mesons include the Pion ($u\bar{d}$) and Kaon ($u\bar{s}$). The bars indicate an antiquark. The leptons carry electric charge and weak isospin and therefore interact via the electromagnetic interaction and the weak interaction. However, the neutrinos do not carry electric charge and so only interact via the weak interaction. The quarks and leptons are arranged into three *generations* in the SM (the three columns in Figure 1.1), where the particles from one generation to the next would be identical if not for the difference in masses.

The SM describes three unified fundamental forces, namely Quantum Electrodynamics (QED), Quantum Chromodynamics (QCD) and the Glashow-Weinberg-Salam model. Processes which interact via the exchange of photons can be described as QED, while those that interact via the exchange of gluons are described as QCD. The Glashow-Weinberg-Salam model unifies electromagnetic and weak interactions into the electroweak theory. Examples of QED, QCD and electroweak processes are electron-muon scattering ($e + \mu \rightarrow e + \mu$), gluon scattering ($g + g \rightarrow u + \bar{u}$) and the decay of the Higgs Boson to a pair of b-quarks ($H \rightarrow b\bar{b}$), respectively.

Although the SM is a very good theory, it is by no means a complete theory as there are questions and phenomena that fall outside of its predictions. A fourth force, gravity, is not very well understood, dark matter is predicted but remains undetected, and there is the unanswered question as to how an imbalance occurred between matter and anti-matter in the creation of our universe. There are SM extensions like Z′ and W′ particles and Beyond Standard Model (BSM) predictions such as supersymmetry and extra dimensions. To further explore such theories experiments have to be performed. The goal is to reduce composite particles to their most simple forms and then compare their behavior to our

models. This is currently being done at the LHC.

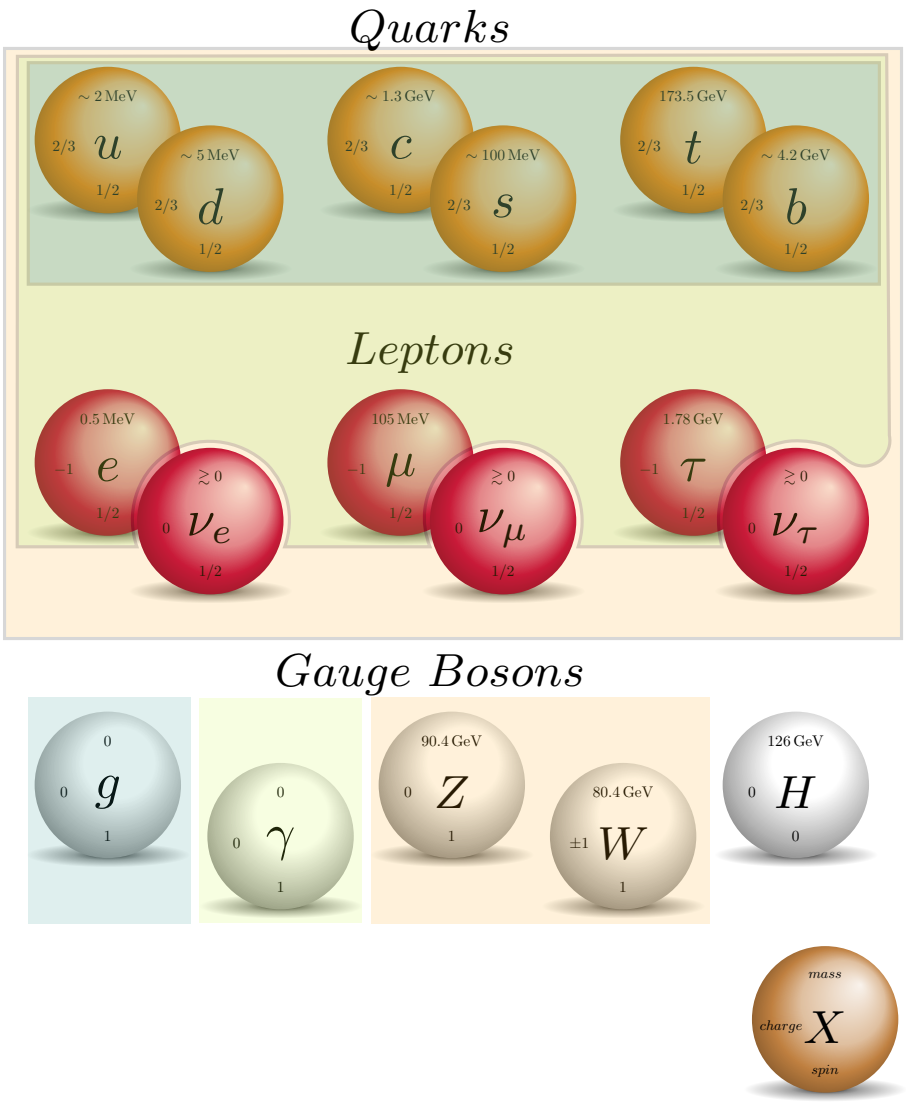


Figure 1.1: The Standard Model of particle physics. Shaded regions indicate possible fermion-boson couplings. The Higgs Boson couples to all massive particles.

1.2 The Large Hadron Collider

The LHC, situated at the European Council for Nuclear Research (CERN), with the location of the four main detectors is shown in Figure 1.2. The beam pipe as well as the detector caverns range from approximately 40 to 180 meters underground. In the case

of proton-proton collisions, the design parameters consist of a luminosity[2] of $10^{34} cm^{-2} s^{-1}$ with an event rate[3] of 40 MHz. With a bunch spacing of 25 ns and around 2800 protons per bunch, this equates to approximately 600 million proton-proton collisions each second with each collision $\mathcal{O}$(MB). This translates to just under 1 PB of raw data being created every second at the ATLAS and CMS IP's.

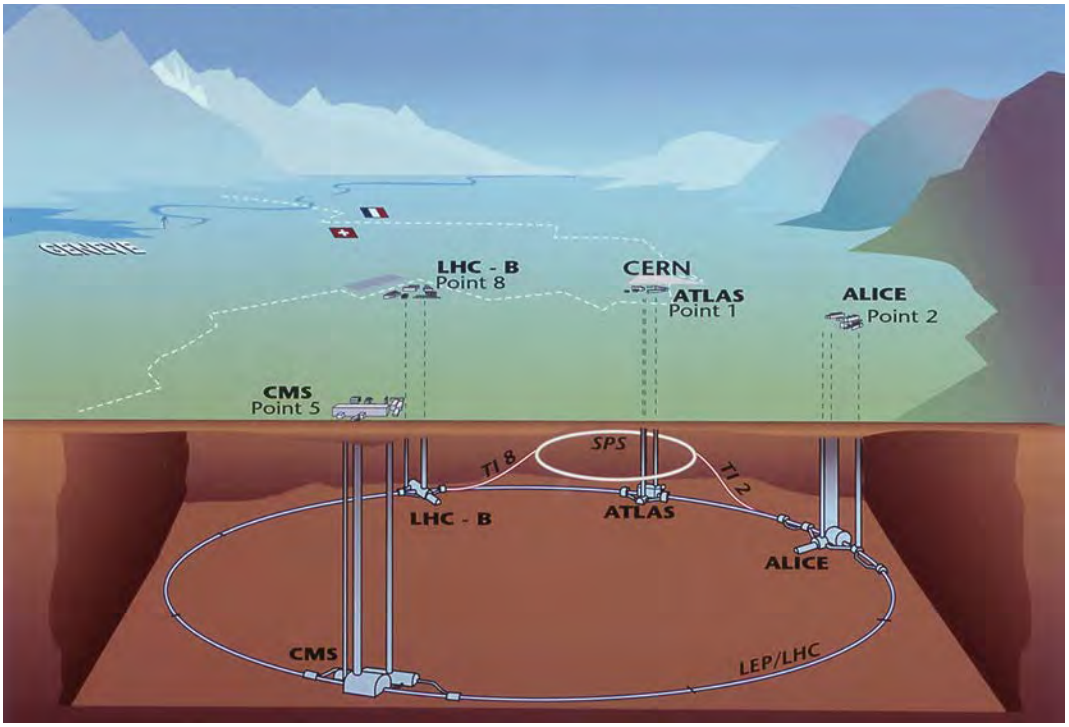


Figure 1.2: The LHC with the location of the four main detectors (image taken from [13]).

Figure 1.3 shows the timeline for the the LHC. The beam was successfully turned on in 2009 and powered down in 2013 for the Long Shutdown 1 (LS1). In this running period the maximum center of mass energy (CME) obtained was 8 TeV, with an integrated luminosity[4] of approximately 26 fb^{-1}. The beam has recently been recommissioned with official data-taking for Run 2 having started on 3 June 2015. There are stable beams at a CME of 13 TeV with a possible increase to 14 TeV in 2016. The initial integrated luminosity will be approximately 75 fb^{-1} increasing to 100 fb^{-1}. An increase in the CME

[2]Luminosity is a measurement of the number of collisions that can be produced in a detector per cm^2 and per second.

[3]An event at the LHC is defined as one bunch of hadrons crossing through another bunch of hadrons from the opposite beam.

[4]The integrated luminosity, measured in fb^{-1}, is a measurement of the number of collisions. Its value characterizes the performance of the accelerator.

and integrated luminosity means that more data are created and thus more computing is needed to analyze them. LS2 is scheduled for the middle of July 2018 until the end of 2019 where the LHC and detector components will be upgraded to accommodate Run 3 with an integrated luminosity of 300 fb^{-1}. Another upgrade period (LS3) will take place to be ready for the high luminosity phase of the project (HL-LHC) where dramatic increases in luminosity and integrated luminosity are planned.

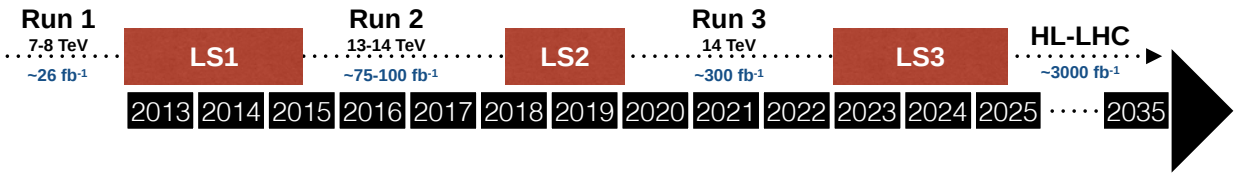


Figure 1.3: The timeline for the LHC experiment.

Clearly, research and development for computing at the LHC needs to be continuous as there is and will be a constant need for higher computing performance. The handling of data can be split into two major types at the LHC; online and offline computing, which will be discussed in the next sections.

1.3 LHC computing

The experiments at CERN need huge amounts of computing power to create, store and move data around for analysis at a later stage. CERN does not have the financial means to provide all computing services on site, so a distributed computing model was developed that incorporates data centers from all over the world. This model includes online and offline computing as well as the LHC Grid, all of which will now be discussed.

1.3.1 Online computing

Online computing is done in real-time and so has to be fast. It comprises all of the data acquisition done at CERN while the beams are switched on. Algorithms decide what data to keep and what to throw away. It is at this point that signals from detectors get converted to raw data through a series of triggers which filter the data down to a manageable size.

Tens of thousands of computer cores work in parallel to reconstruct full events and even then only approximately 0.005% of these events are kept for later analysis. It is up to the implemented algorithms to make the decision of what constitutes an interesting event.

Another aspect that makes up online computing is that of high-bandwidth networking. Different instruments placed around the detectors need to communicate with each other in order to reconstruct events. This requires a high speed communication line $\mathcal{O}$(TB/s). Technologies such as PCI Express and InifiniBand are options that may produce the required results but need to be explored further.

With the startup of the LHC at higher collision energies, the amount of data created will only increase, thus straining current technologies implemented in the different detectors. LS1 has enabled detector components as well as algorithms to be updated to be able to cope with this new influx of data. While online computing is obviously important, and some ideas and topics in the following chapters can be applied to online computing, it will not be the focus of this thesis. Rather, offline computing will be explored.

1.3.2 Offline computing

Offline computing incorporates every type of computer operation that is done once the online computing components have collected and stored the data. Offline computing has fewer risks involved in that data can not be permanently destroyed. Analyses can be repeatedly run, albeit at a cost of the user's time.

There are different types of data analysis. For example weather pattern predictions, fluid dynamics and molecular biology, all require specialized super computers built and programmed for specific tasks. This is called High Performance Computing (HPC), where the goal is to build the fastest computer possible, comprised of state-of-the-art hardware all on one site. Some disadvantages to this type of data analysis is the cost required in building such a computer as well as scalability. Although it is possible to increase the overall processing power by adding more cores, for these types of problems this isn't necessarily helpful. Another type of data analysis (that which applies to the LHC and HEP in general) is called High Throughput Computing (HTC), where commodity hardware computing clusters are placed in different physical locations, but are in constant communication with each other. The goal is to run as many independent calculations as fast as possible and thus have a higher throughput of data. The advantage to HTC is that it is not location specific. Data analysis for one user can be carried out in South Africa and Geneva si-

multaneously without any drawbacks. Cost and scalability are also advantages to HTC. It is up to the participating institute to determine how large a server farm they need, and how much they can afford. Having few cores is acceptable but having more cores in HTC is never a disadvantage. This type of data analysis is called *embarrassingly parallel*.[5] In order for this type of data analysis to work, an extensive framework and network must be used to coordinate each server farm and share resources.

1.3.3 The LHC Grid

The LHC Grid was designed and built to handle the distributed computing demands of the LHC. In Run 1, from 2010 to 2012, around 25 PB of data were created each year. This data has to be accessible to different data centers around the world. Users can run "jobs" from any geographic location using the resources of the LHC Grid. Hundreds of thousands of jobs run everyday. Figure 1.4 shows the distribution of server farms. While most are in Europe, every continent except Antarctica has at least one server farm connected to the LHC Grid.

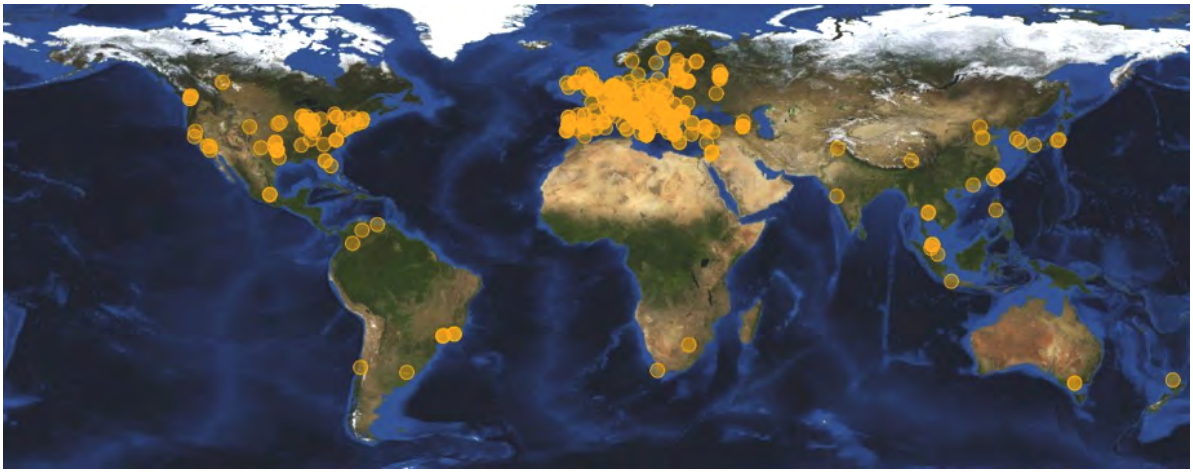


Figure 1.4: The distribution of the LHC Grid server farms across the world (image taken from [14]).

There is a hierarchy of "Tiers" in the LHC Grid where each Tier has a function to perform. These are as follows:

- Tier 0 is the largest and most powerful of the data centers. Until recently, there

[5] A real computing term highlighting the apparent ease at which the problem can be parallelized.

was only one Tier 0 located at CERN, however, a server farm in Hungary has also acquired Tier 0 status. All data created by the LHC passes through these centers. Storage, reconstruction and the distribution of these data to Tier 1 centers are the foremost responsibilities. In early 2013, CERN's Tier 0 had a power capacity of 3.5 MW.

- Tier 1 centers retrieve the data from CERN and Hungary. These are 13 large data centers responsible for simulation, reconstruction and storing a portion of the raw data. They then redistribute this data to Tier 2 centers.
- Tier 2 centers are located at universities and institutions around the world where they generally have less disk space and fewer CPU's. There are around 160 of these centers. This is where a user will run their analysis. They are also assigned a portion of simulated event production and reconstruction. Of special interest is the South African Center for High Performance Computing (CHPC) which has recently become an official member of the LHC Grid and will host a Tier 2.
- Tier 3 centers are not connected to the LHC Grid and therefore are not assigned a proportional share of tasks. They are often smaller, consisting from a single computer up to a few servers providing access to only local users.

1.3.4 The ROOT framework

HEP analysis makes use of ROOT [15] which is a framework for performing large scale data analysis. It was first developed in the 1990's as PAW (Physics Analysis Workbook), but it was realized that the older Fortran libraries would not be able to keep up with the amount of data that the LHC was projected to create. Thus, ROOT was written in C++. More recently it has the capability of adding python bindings which makes it PyROOT [16]. ROOT is an Object-Orientated program which offers data and event reconstruction, data analysis, detector simulation and tracking. The authors emphasize the word "framework". This is because ROOT was written specifically for HEP which includes physics objects, histograms and fitting. The advantage to using this prescribed framework is that everything one would need for a complete physics analysis should be there, thus simplifying the process of writing code. ROOT has become the primary choice for the HEP community.

1.3.5 PROOF

ROOT makes use of sequential data analysis and so the obvious next step was to see how well it can be parallelized, thus the extension PROOF [17] was created. PROOF strives to introduce maximum parallelism by splitting up files among workers. A master node (or core) distributes the files to all the different CPU's that it is able to access. This could be the CPU's on it's own board or those located on other boards connected within the same network. Thus, where one computer with eight cores running ROOT could only analyze one file at a time, if PROOF is enabled eight files can be analyzed simultaneously. PROOF is meant as an alternative to batch systems such as those implemented in the LHC Grid. It can be run interactively and so is ideal for smaller clusters at institutions that are not connected to the grid.

It is important to note that there are other ways one can run PROOF. PROOF-Lite is a simple implementation of PROOF and should work with no extra setup required on any machine that has ROOT. The difference is that PROOF-Lite only makes use of the extra cores available on a single computer and cannot be scaled up. Another implementation of PROOF is Proof-On-Demand (PoD) [18]. This is a version that has large potential for a cloud computing environment. At the request of a user, PROOF can be set up and an analysis carried out on a cluster that requires no administration rights. It is portable and has been written with the aim of making it user friendly.

There are disadvantages to using PROOF. A typical user is required to have a deeper knowledge of ROOT. Most users would rather submit their jobs to a batch system that only makes use of CPU parallelism and wait that much longer, than have to learn the intricacies of PROOF. Furthermore, a well written batch system that makes use of core parallelism (such as the example in the E/p analysis) may turn out to be faster as there is very little communication overhead. Another downfall is the difficulty of setting up a dedicated PROOF cluster and the resources required to do so. Using an NFS directory drastically simplifies things as this prevents one having to setup XRootD configuration files, but it impedes performance, (Chapter 4.5).

1.4 The ATLAS experiment

ATLAS (A Toroidal LHC ApparatuS) is a general purpose particle detector designed to study SM and BSM physics. The detector stands 25 meters tall, 44 meters long and

weighs approximately 7000 tonnes, (Figure 1.5).

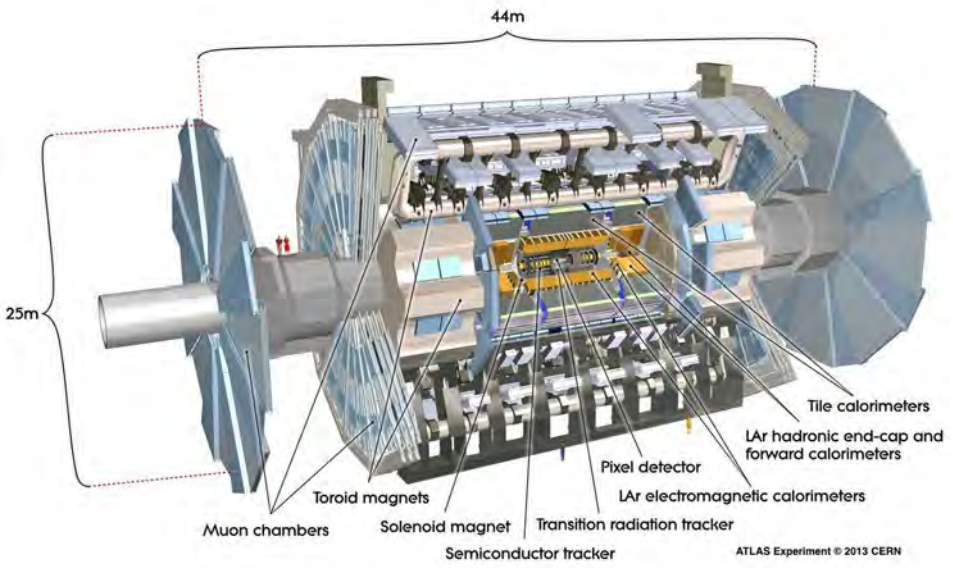


Figure 1.5: The ATLAS detector with labelled components (image taken from [5]).

There are 4 main hardware components that contribute to the detection of the particles produced in the collisions created by the LHC. These are as follows:

- The Inner Detector (ID) is approximately 6 m long with an outer radius of 1 m. It consists of three independent components. From closest proximity to the beryllium beam-pipe, these are: the pixel layer, the SemiConductor Tracker (SCT) and the Transition Radiation Tracker (TRT). These are shown in Figure 1.6. The pixel layer and SCT are similar in that they are both silicon based detectors. The pixel layer is made up of smaller discrete modules or "pixels", while the SCT contains longer strips of silicon. Therefore, the SCT is capable of covering more area but has a coarser granularity. The TRT is comprised of drift tubes, each filled with a $Xe/CO_2/O_2$ (70%,27%,3%) gas mixture. The gas becomes ionized when charged particles pass through, while a wire running through the center of each tube detects small signals due to the negative ions produced during ionization. The ID is responsible for high precision measurements of charged particles. The pixel layer, SCT and TRT combined have over 86 million readout channels.

- The ATLAS calorimeter system is comprised of electromagnetic (EM) and hadronic calorimeters, shown in Figure 1.7. Electrons and photons can be measured with

high precision due to the fine granularity of the EM calorimeter. It is comprised of the liquid Argon (LAr) electromagnetic barrel and end-caps, which contain lead and stainless steel to absorb the particle's energy as well as LAr as the active material. The hadronic calorimeters are comprised of the Tile Calorimeter (TileCal), the LAr hadronic end-caps and the LAr forward calorimeter. The coarser granularity of the hadronic calorimeters makes them more suitable for reconstruction of jets and missing transverse energy.[6] The TileCal uses a scintillator plastic as the active material, and iron as the absorber. It consists of three main cylinders; the long barrel and two extended barrels. The TileCal will be further discussed in Chapter 4.4.

- The Muon Spectrometer (MS) covers the largest volume in the ATLAS detector. Its function is to measure the properties of the muons that penetrate through all the other detector components. Precision measurements of the momentum are taken using the Monitored Drift Tube chambers (MDT's). These are comprised of several layers of tubes, each filed with a Ar/CO_2 (93%,7%) mixture of gas. The Resistive plate chambers (RPC's) are filled with a $C_2H_2F_4/Iso-C_4H_{10}/SF_6$ (94.7%,5%,0.3%) mixture of gas. Their data acquisition rate is fast enough that they are suitable for use in the trigger decision. These components are shown in Figure 1.8.

- The magnet systems in the ATLAS detector are 26 m long and 22 m wide. Figure 1.9 shows the geometric setup of the different magnet components, along with the TileCal. The 2 T central solenoid is aligned along the beam axis, surrounding the ID and encompassed by the TileCal. This curves the particles and thus allows for their momentum to be determined. The barrel and end-cap toroidal magnets provide a field that ranges between 0.5 and 1 T and similarly allows for the identification of charge and momentum due to the particle's curvature.

These detector components create $\mathcal{O}$(PB/s) of raw data. This needs to be reduced so that storage systems can cope with the volume. The first level of the trigger system (L1) is hardware based and selects events with large energy deposits and hits in the calorimeters and muon spectrometers, respectively. The larger granularities of these detectors enable fast decisions. The next level is the high level trigger (HLT) which is comprised of the L2 trigger and the Event Filter (EF). These are software based and run in real time in data centers that are physically close to the detectors. Being physically close to the detector reduces the time for data packets to be sent and received. The L2 does a fast

[6] Missing transverse energy is energy carried away by a neutrino in an event. None of the detectors at the LHC can directly detect neutrinos due to their property of interacting very weakly with matter.

reconstruction of the event, where all the detector information within the region of interest is used. The EF then performs a further reduction in the event size. At this point all the event information is available and more thorough algorithms decide whether or not the event is of further interest for offline analysis.

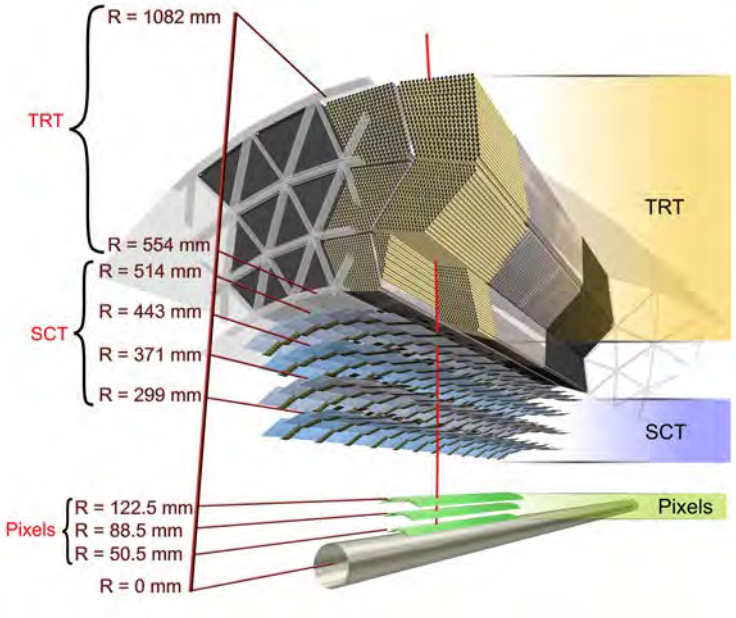


Figure 1.6: The ATLAS Inner Detector showing a charged track (in red) traversing the beam-pipe, the silicon pixel layers, the SCT and the TRT (image taken from [5]).

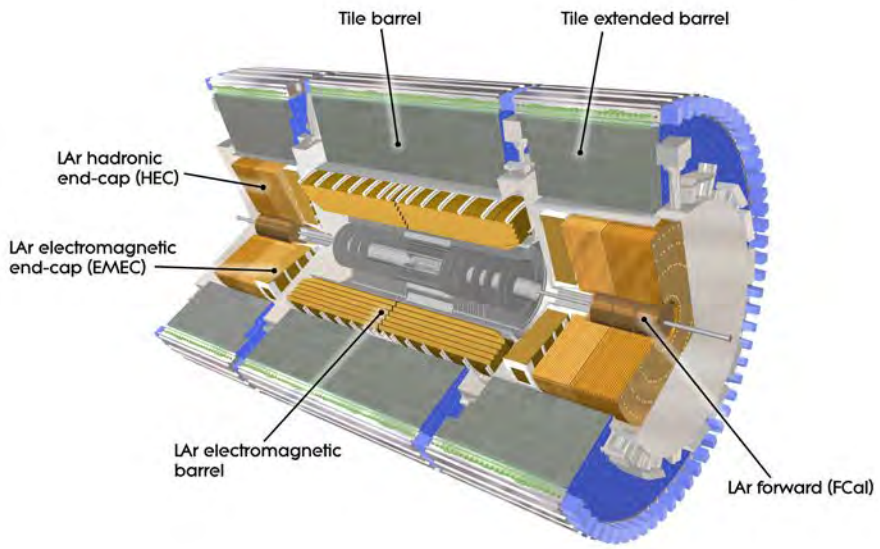


Figure 1.7: The ATLAS electromagnetic and hadronic calorimeter systems (image taken from [5]).

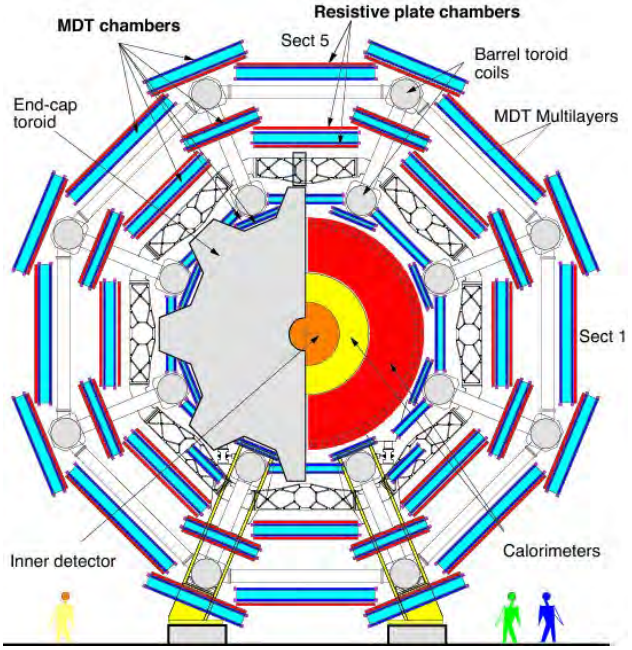


Figure 1.8: A transverse slice of the ATLAS detector showing the Muon Spectrometer components (image taken from [19]).

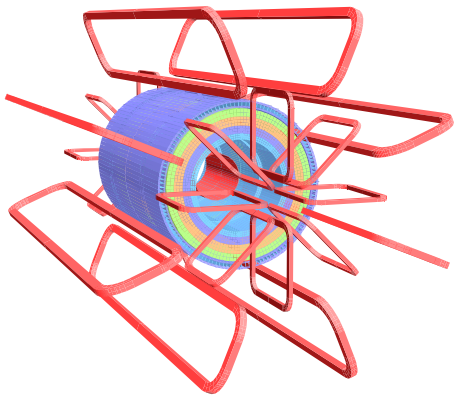


Figure 1.9: The geometry of the ATLAS magnet systems and the TileCal without the other detector components (image taken from [5]).

1.4.1 Data parallelism with ATLAS

The online systems (such as the detector components mentioned above) all share a common theme in that data acquisition is done discretely. Each collision in the detector is essentially a separate stream of data. This means that recording signals and converting them into data can be carried out simultaneously on many different detector components and computers. The same theme can be applied to offline computing at ATLAS and the LHC in general.

The detector components create and store the data in a readable format called a dataset. Each dataset is composed of N number of files, each filled with x number of events. A schematic is shown in Figure 1.10. A data analysis at CERN involves performing operations on any number of events inside files which reside inside datasets. If D datasets are used then cores can be assigned separate datasets to work on. Thus, each core has $D \times N$ files and $D \times N \times x$ events. This is now an embarrassingly parallel problem. All cores run the same code, are memory safe and need very little communication between each other. Parallelism at the event level is often not done as it is more difficult and requires writing the analysis specifically for that purpose. Parallelism at the dataset and file level just requires a general wrapper that will split up the files then start jobs on each core. The nature of parallelism will be further explored in Chapter 2. This style of parallelism will be referred to as *batch computing* with an example of this is shown in the E/p analysis benchmark presented in Section 4.4.

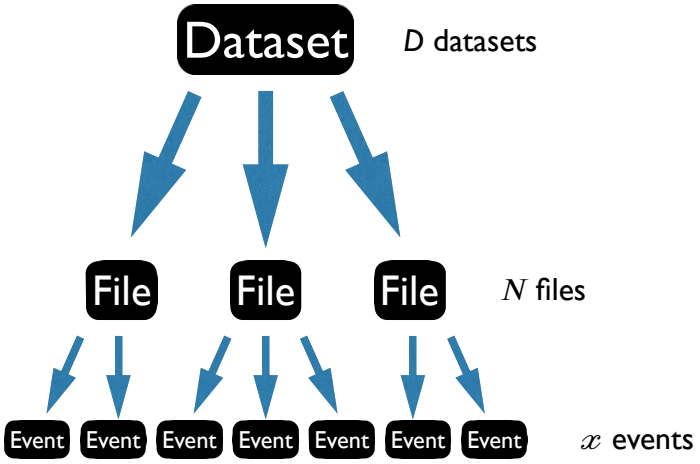


Figure 1.10: The hierarchy of how data from CERN can be split up to be analyzed.

1.5 Other main LHC experiments

The CMS (Compact Muon Solenoid) detector stands 15 meters tall, 22 meters long and weighs approximately 14000 tonnes. It is another general purpose detector with the same physics objectives as ATLAS. However their detector components and thus detection of particles is different to ATLAS but with comparable performance. This way the collaborations can cross check their results.

ALICE (A Large Ion Collider Experiment) [20] is a specialized detector with the specific goal of studying the properties of the quark-gluon plasma (QGP).[7] This is a state of matter where confinement rules no longer apply and quarks and gluons move freely in this "perfect liquid". It is believed that these are the conditions that existed very briefly after the Big Bang. At a predetermined time in an LHC run, the proton beams are switched to lead beams. Thus, ALICE is able to make use of the higher nucleus densities in lead-lead collisions, which provide optimal conditions for a QGP.

LHCb (LHC beauty) [21] is another specialized detector. It's main objective is to study the decay of particles containing "b-flavoured" quarks. One of the properties of these particles is that they stay close to the beam line. Thus, the LHCb detector is asymmetric. It surrounds the beam pipe for 20 meters in one direction weighing approximately 4500 tonnes.

1.6 A note on the Square Kilometer Array

It seems prudent that the Square Kilometer Array (SKA) is briefly mentioned since it will also be dealing with data on an enormous scale. As such, the implications of this thesis are applicable to the SKA. The SKA is another global scientific experiment, which at this moment consists of 11 member countries with telescopes located in South Africa and Australia. It will be the largest radio telescope in the world.

The project timeline has been split into two phases. In the first phase topics such as the understanding of neutral hydrogen in the Universe, and testing the theories of general relativity and quantum gravity will be explored, as well as many more interesting topics. Eventually, the SKA will consist of millions of antennas and thousands of satellite dishes

[7] Quarks come in different flavours and colours and make up 6 out of 12 of the fundamental particles. Gluons are fundamental force carries.

providing astronomers with $\mathcal{O}$(PB/s) of data. Parallelism with the SKA is not as trivial as it is for the LHC because it is not straightforward to reduce the volume of the data created. This is a significant challenge in the equivalent online and offline systems. It is estimated that the computing requirement needed for various applications will be on the order of *exa-flops*, [22] that is 1×10^{18} floating operations per second. According to the TOP500 [23] list from November 2014, the fastest supercomputer is capable of reaching 33.86×10^{15} *flops*. The power draw of this supercomputer, including the crucial external cooling, is 24 MW. The data from the SKA has to be processed on site as the technology to move data is not fast enough. With the scale of processing power needed, this means supplying huge amounts of power to remote areas in South Africa, a country already experiencing severe power shortages. Novel techniques and hardware will need to be developed so that computing performance per watt increases. Thus, some work done in this thesis will be of interest to the SKA.

Chapter 2

Distributed and Parallel Computing

One of the most common misconceptions about parallel computing is that if you use more CPU's in an application it will require less time to complete the task. While this can be true, in most data analysis or scientific applications it is incorrect. There are many factors that will impair a computers' ability to parallelize a task or force it to serialize the task altogether. Shared memory between the CPU's creates bottlenecks, communication between the CPU's when sending and receiving information, and latency when starting processors will all contribute to overhead. Above all, there are two crucial elements to parallelizing a task. The first is the nature of the problem. Some scientific calculations or tasks are just not possible to effectively parallelize, while others will benefit greatly. The second is the coders' ability and skill in parallelizing code. This latter issue represents a different challenge which will not be further discussed in this thesis. The nature of the problem and what parallelism actually is will be explored.

2.1 The nature of parallelism

There are two main types of parallelism. The first is when an infinite number of threads can be spawned and similar tasks are given to each thread.[1] The threads all share the same properties (such as variables) and in the case of Python, the same interpreter. This can have disadvantages and advantages. If multiple threads try to access the same variables, severe consequences can occur, where in the best case running the script will

[1] Obviously, at some point too many threads will hamper performance, but the point is that any number of threads can be specified.

fail, in the worst case it will carry on giving the wrong results. The Python interpreter only allows single instructions to be carried out at any given time, and so, when using threads there is still the possibility that large fractions of code will be carried out sequentially. Another disadvantage is that using threads limits the code to one computer (and however many cores that computer has). Communication across computers is not possible and so ultimately this type of parallelism is not scalable. The advantage to using threads is that only small changes to the code are required for it to be parallelized. Furthermore, different programming languages have modules with documentation that can be used to exploit thread-level parallelism.

The other approach to parallelism is to use separate processors to spawn different processes.[2] This means that each process is memory safe and that communication across physical computers is possible. Thus, this type of parallelism is highly scalable. The disadvantage to this method is that overhead increases as data objects have to be sent and received to and from different computers. Cluster topologies and communication now has to be considered. For this reason, this method requires more adjustments to the code as consideration has to be given as to how and from where the data is read, and how to package it and send it back to the host CPU.

Some examples are shown to illustrate the above. One of the most basic pieces of code that can be parallelized is a "for-loop" shown in Listing 2.1. One approach would be to spawn threads and take advantage of the simplicity of the code. As soon as a thread finishes the task it grabs the next number in sequence. There is little overhead involved, so this code should see great improvement when parallelized. However, at some number of threads a point of saturation will occur and no more increase in speed will be recorded. The other approach is to split the domain of iteration between n processors where each will run a process. Thought has to be given as to how to send and retrieve the data and so the need for communication will be greater in this case. If the time taken for the task to be performed is much larger than the overhead, then the parallel time of computation will show improvement to that of the sequential time. If the time taken for the task to be performed is on the same order of the overhead, then communication and latency will dominate the process. Thus, the time to perform the task in parallel could become slower than the sequential time.

The approaches to parallelization mentioned above can be found in already bundled software packages. The application programming interface (API) called Open Multi-Processing

[2] Note that a processor is a physical piece of hardware and a process is what can be spawned on threads or processors.

Listing 2.1: A basic "for-loop" depicting the best case scenario for parallelizing code.

```
Products=[]
for i in range(1,10000):
      Products.append(i*10)
print Products
```

(OpenMP) [24] supports intra-node parallelization between cores sharing memory. Thus, OpenMP cannot communicate across nodes. The API called Open Message Passing Interface (Open MPI) [25] offers the ability to communicate across nodes and so supports inter-node parallelization; it is therefore scalable. There are other language specific modules and APIs available that allow for inter and intra-node parallelization. However from here on when the terms OpenMP and Open MPI are used they refer to the strategies needed when coding for intra- and inter-nodal networks, respectively.

Listing 2.2 shows an example of a "for-loop" where one would think this has a larger portion of the code with potential for parallelization. However, it turns out that the outer loop is the better choice to parallelize, leaving the inner loop as is. The added overhead combined with a dependence on the outer loop makes the inner loop difficult to parallelize effectively. It is possible to use a combination of OpenMP and Open MPI to achieve faster computation time, however this is a non-trivial task.

Listing 2.2: Two basic "for-loops" depicting a case where it looks as if a larger fraction of the code is parallelizable.

```
Products=[]
for i in range(1,10000):
   for j in range(100,200):
      Products.append(i*j)
print Products
```

Listing 2.3 is an example that would be difficult to parallelize using OpenMP due to the dependence on variables that are constantly being read and rewritten. Suppose we have one thread calculating the case where $i = 2$ and another thread calculating for $i = 3$ simultaneously. We have $C[2] = A[2] + B[3] + C[1]$ and $C[3] = A[3] + B[4] + C[2]$, respectively. The second equation calls the element $C[2]$ while the first equation is still updating the list. Whether or not the first equation will update the element before the second equation reads from it is unknown and so can give incorrect answers. The solution lies in providing locks to certain variables ensuring that they can only be accessed by one thread at a

time. This serializes the code. The other approach is to use Open MPI by splitting up the domains of each list and sending them to new processors where no overlap will occur. This requires large changes to the code and increases overall overhead. In this example the calculation is not intensive and so carrying out this task sequentially will be quicker than implementing Open MPI.

Listing 2.3: A "for-loop" that is difficult to parallelize due to the dependence on a variable that needs to be read in order. The "..." represents lines of code that would be needed to handle the boundary indexing conditions.

```
A=[...]
B=[...]
c=[...]
for i in range(len(A)):
   ...
   C[i]=A[i]+B[i+1]+C[i-1]
   ...
   print C[i]
```

These examples provide a general feel for how tasks can be parallelized and the challenges that appear.

2.2 Performance models

This section presents some basic, yet insightful performance models that describe parallel systems. The two models that will be discussed focus on tasks of a *fixed size* and tasks of a *fixed time*. Examples of each for the LHC context are as follows:

- *Fixed size model*: In this model the problem size parameter does not change. This can apply to an analysis done at the LHC. If a group is looking at $H \rightarrow b\bar{b}$ using the 2012 dataset, the number of data does not change. There will always be $D \times N \times x$ events that need to be used in the analysis (refer to Figure 1.10).
- *Fixed time model*: In this model the problem time parameter does not change. At the LHC, this can apply to data acquisition. In this case, the fixed time model is limited by the offline computing components. The online components and software in the detectors reduce a 40 MHz event rate down to approximately 1kHz so that current I/O and CPU technologies can handle the data. An example applies to the LHC grid.

Everyday, hundreds of thousands of jobs are run at different server farms located around the world. At some point the LHC grid reaches a point of saturation where new jobs submitted will have to be queued. If more CPU's are to be added to the server farms, jobs could be run more quickly and more data can be analyzed. Thus, theoretically with infinite CPU's, infinite jobs can be run at the same time.

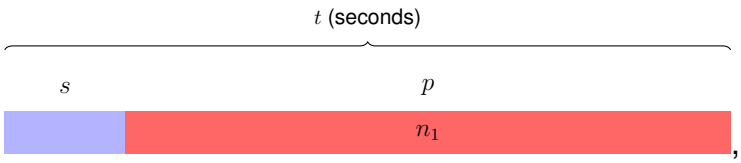
It is important to note that simulation of events can be modeled as either fixed time or fixed size model. The terms *speedup*, *relative speedup* and *absolute speedup* also need to be defined:

- *Speedup* - This is a term used to define the increased or decreased performance of task that is carried out multiple times on n number of processors. It may be unitless or have units of time.
- *Relative speedup*: This is the speedup of the task when compared to the sequential performance done on one processor of that same computer. We will assume that the code has been optimized to perform at it's fastest possible speed for both sequential and parallel computations. For example, if we multiply two $n \times n$ matrices sequentially on one core of the multicore processor it will take some time to complete depending on the the type of processor used. If we now parallelize our code and run it on four of the cores of the same processor then the amount of time taken for the task will decrease. Thus, there will be a speedup using four cores relative to the time taken on one core for the same processor.
- *Absolute speedup*: This is a metric that can help to determine whether or not it's worthwhile replacing traditional processors with more, cheaper and slower processors. For example, if we multiply two $n \times n$ matrices we can measure the time it takes to complete the task on the fastest processor available. This doesn't have to be a sequential task, in fact it probably won't be. We then measure the time taken to complete the task on the cheaper processor (also in parallel). The ratio of these two times is defined as absolute speedup. The questions that arise from this are:
  - How many cheap and slower processors are needed before absolute speedup starts favoring the slower processors?
  - If these cheaper and slower processors use less power, will the overall configuration still use less power and perform at faster speeds?

2.2.1 Amdahl's Law (fixed size model)

Amdahl's Law [26] says that there is a finite cap on the relative speedup for a problem of fixed size, regardless of how many cores that are assigned to the problem. The relative speedup is therefore limited by the part of the code that has to run sequentially. The time to the solution may vary but the problem size, as an assumption of the model, will not.

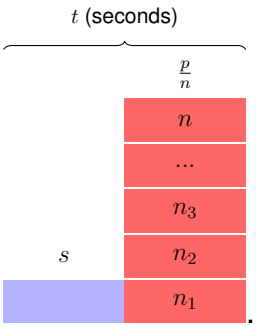
If we assign s to denote the fraction of the task's runtime (t) that has to be carried out on a single core (i.e sequentially) and p to be the fraction of the task's runtime that has the potential to be carried out in parallel, we can visually show this in a *thread diagram* as



where n_i refers to number of workers. These diagrams help give an intuitive sense of the task at hand. From this we can say that

$$s + p = 1. \tag{2.1}$$

The parallelizable portion of the code can be split up among n workers, thus the *thread diagram* becomes



From these two thread diagrams we can say that

$$T(n) = (s + \frac{p}{n})t, \tag{2.2}$$

where in the case $n = 1$ we have $T(1) = t$.

Performance can be defined as

$$\psi(n) = \frac{N}{T(n)}, \tag{2.3}$$

where N is the fixed problem size. The units of performance are computations per second, where computations are defined by files, events, megabytes et cetera.[3] From this we can obtain relative speedup for the fixed size model (S_R^{fs}), which is the ratio of performance of the computer running the task in parallel to the same machine running the task on one worker:

$$S_R^{fs}(n) = \frac{\psi(n)}{\psi(1)} \tag{2.4}$$

$$= \frac{T(1)}{T(n)}. \tag{2.5}$$

Using Equations 2.1 and 2.2, Equation 2.5 can be simplified to

$$S_R^{fs}(n) = \frac{1}{s + \frac{1-s}{n}}. \tag{2.6}$$

This is Amdahl's Law and is plotted as a function of n in Figure 2.1. Notice that as $n \to \infty$, relative speedup goes as $1/s$. Clearly, the downfall to parallelizing code for a problem of fixed size is that only a tiny portion of it can be sequential for there to be worthwhile relative speedup.

The efficiency (ξ) of each added worker can be defined as

$$\xi(n) = \frac{S_R^{fs}(n)}{n}, \tag{2.7}$$

which is also shown in Figure 2.1. A simple example for a fixed size problem is multiplying two $n \times n$ matrices together. If one person were to do it without a computer it would take some time, t. If n people were to do it, they could split it up accordingly, maybe assign one person one line, up to n. In an ideal world the time for this computation is now t/n. However should there be an amount of people $> n$, there isn't more work and so they stand by idly watching.

The absolute speedup can be obtained (S_A) similarly by defining performance for "fast"

[3] The performance metric does not have to be defined, but it provides another way of calculating speedup which is used in Chapter 4.3.

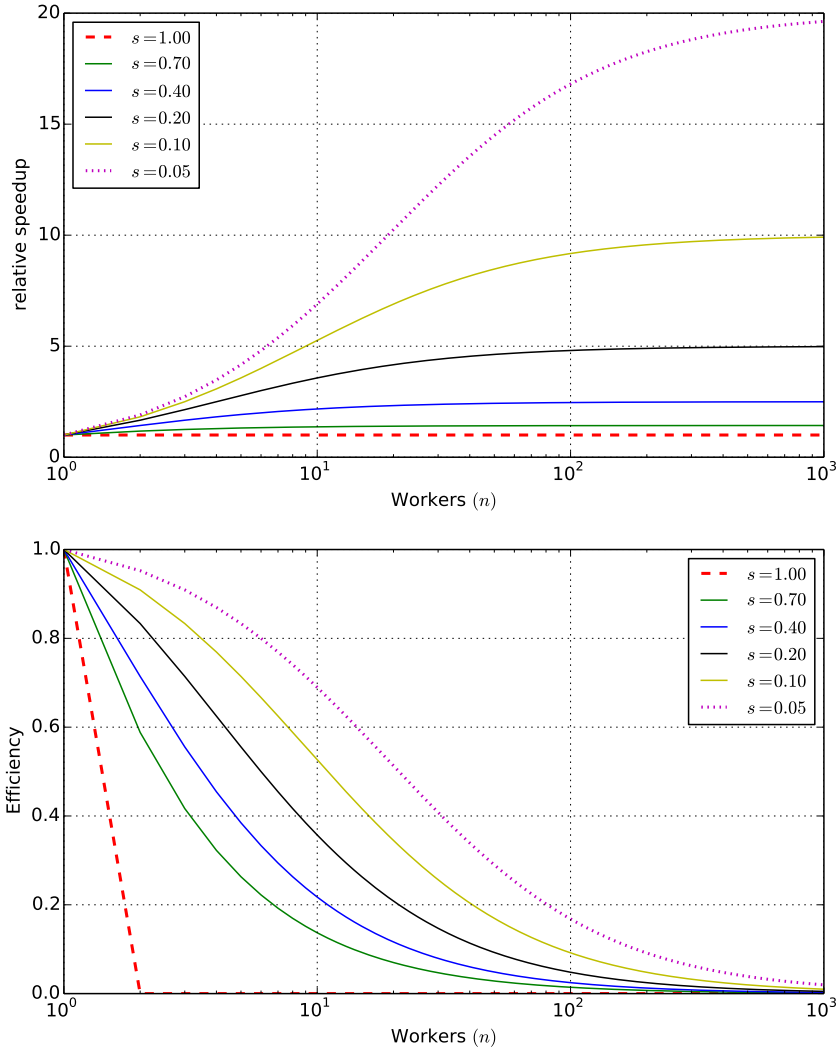


Figure 2.1: Top: Relative speedup for fixed size model. Bottom: Efficiency for fixed size model. s denotes the fraction of code that is sequential.

(ψ_{fast}) and "slow" (ψ_{slow}) processors so that

$$S_A(n) = \frac{\psi_{fast}(n)}{\psi_{slow}(n)} = \frac{T_{slow}(n)}{T_{fast}(n)}. \tag{2.8}$$

At times $1/S_A$ may be quoted instead of S_A. This is just a choice to maintain consistency in the derivation and thus, quote the more intuitive result as a fraction < 1.

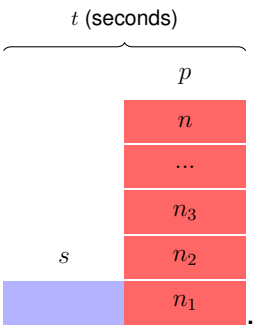
2.2.2 Gustafson's Law (fixed time model)

Gustation's law [27] was introduced in 1988 and applies to a model where the overall problem time is fixed. The number of processors and thus, the problem size can increase. It says that if four cores carry out a task of size four (arbitrary units), it should take the same amount of time for fifty cores to carry out a task of size fifty.

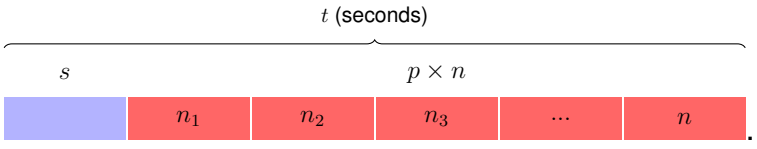
According to [28], relative speedup for the fixed time model (S_R^{ft}) can be defined as

$$S_R^{ft} = \frac{\text{Sequential time of solving scaled problem}}{\text{Parallel time of solving scaled problem}} \tag{2.9}$$

It is useful to draw the thread diagram for the scaled problem size with the code running in parallel first



Note that the parallel portion of the code doesn't have the $\frac{1}{n}$ factor as it did for the fixed size case. This is because for the fixed time model, the parallel runtime will always be the same; five cores running five tasks is the same as ten cores running ten tasks. The thread diagram for the scaled problem size with code being run sequentially can be shown as



Thus, relative speedup for the fixed time model becomes

$$S_R^{ft}(n) = \frac{(s + p \times n)t}{(s + p)t} = s + n(1 - s). \tag{2.10}$$

This is Gustafson's Law, which says that speedup is linear with the number of cores,

provided that the problem size is scaled up to maintain a constant execution time. This is plotted in Figure 2.2.

Similarly to Equation 2.7, the efficiency can be be defined as

$$\xi(n) = \frac{S_R^{ft}(n)}{n}, \tag{2.11}$$

which is also plotted in Figure 2.2.

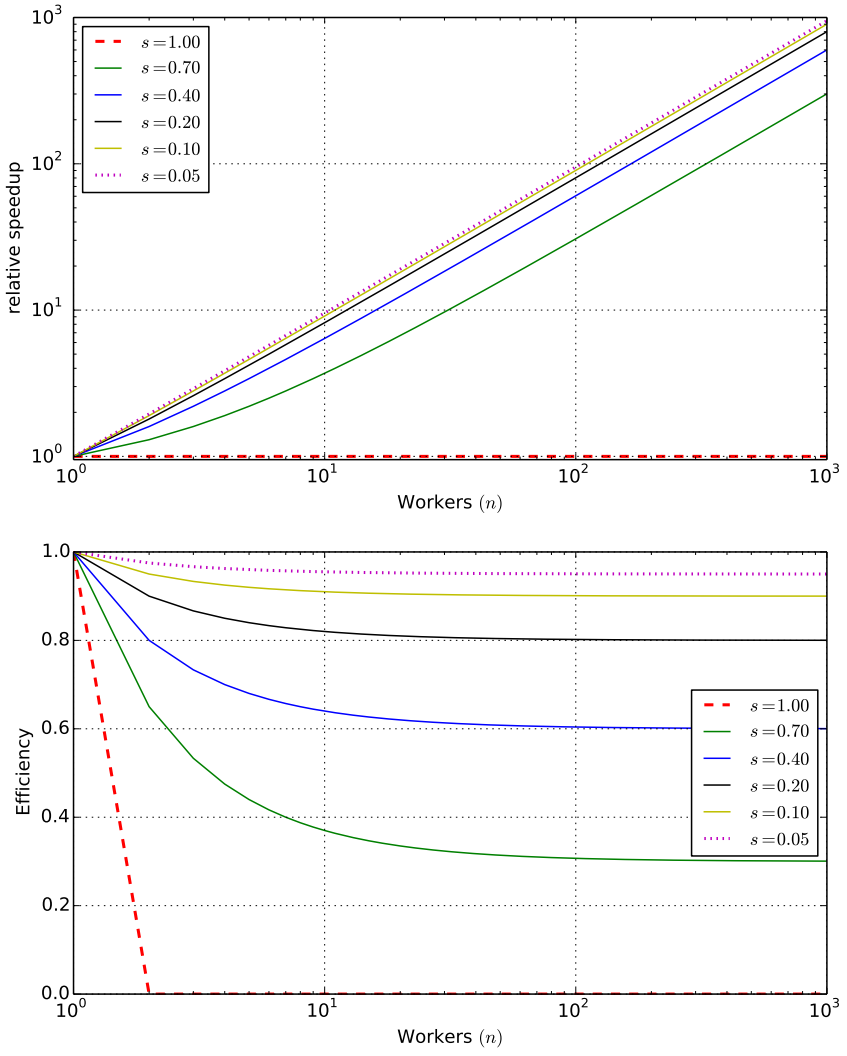


Figure 2.2: Top: Relative speedup for fixed time model. Bottom: Efficiency for fixed time model.

Both Amdahl's and Gustafson's Laws do not specifically take communication into account.

However, as CPU speeds increase communication between cores and different computers have a larger impact. It is therefore worthwhile to see what affect communication has on speedup. It needs to be noted that it is more complicated incorporating forms of communication into Gustafson's Law due to the nature of the time parameter being fixed. Thus, since it does not re-appear in this thesis it will not be shown in the coming sections.

2.2.3 Fixed size model with communication

A basic performance model for the fixed sized model will now be developed that takes into account communication. This can be parameterized by latency (L) and streaming (σ) which are defined as:

- Latency: The elapsed time between a command sent from the host and the first instance of that task being initiated. This is an internal and external parameter. In other words, latency can occur between separate CPU's as well as between the cores of one CPU.
- Streaming: The elapsed time for a message of varying size to be sent. This applies to messages sent to separate CPU's as well as within individual CPU's. It is given by $\sigma = N_{ms}/B$ where N_{ms} is the transferred message size and B is the bandwidth.

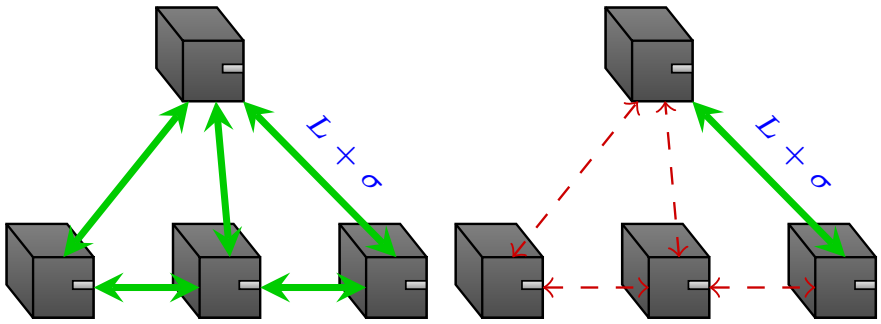
These new variables make up an new parameter called *overhead*, $\omega = L + \sigma$. In the case where the task is run on a single worker, overhead is normalized to one. There are two important scenarios for overhead when looking to parallelize a system. A real-world application will fall somewhere between the two. These examples can apply to intra- and inter- nodal networks.

- Embarrassingly parallel, non-blocking network: In this case files are split among processors and it's assumed that the network can maintain any amount of communication at once. Thus, messages can be sent from the master node to any worker node as shown in Figure 2.3(a). Our overhead is simply $\omega = (L + \sigma)$ and relative speedup becomes

$$S_R^{fs}(n) = \frac{1}{s + \frac{1-s}{n} + \frac{\omega}{t(1)}}. \tag{2.12}$$

- Embarrassingly parallel, blocking network: In this case the task can be split up among the additional processors but if there isn't a suitable communication network the task stalls and messages have to be sent sequentially. This is shown in Figure 2.3(b). Thus, overhead can be written as $n^2\omega = n^2(L + \sigma)$ and relative speedup becomes

$$S_R^{fs}(n) = \frac{1}{s + \frac{1-s}{n} + \frac{n^2\omega}{t(1)}}. \tag{2.13}$$



(a) Non-blocking communication (b) Blocking communication

Figure 2.3: The two basic and extreme cases for network communication.

Figure 2.4 shows the differences for embarrassingly parallel blocking and non-blocking networks for the fixed size model with values of $s = 0.05$ and $\omega = 0.002$. Notice for the blocking network, if more processors are added beyond a point of saturation, speedup decreases to the point where it is less than one.

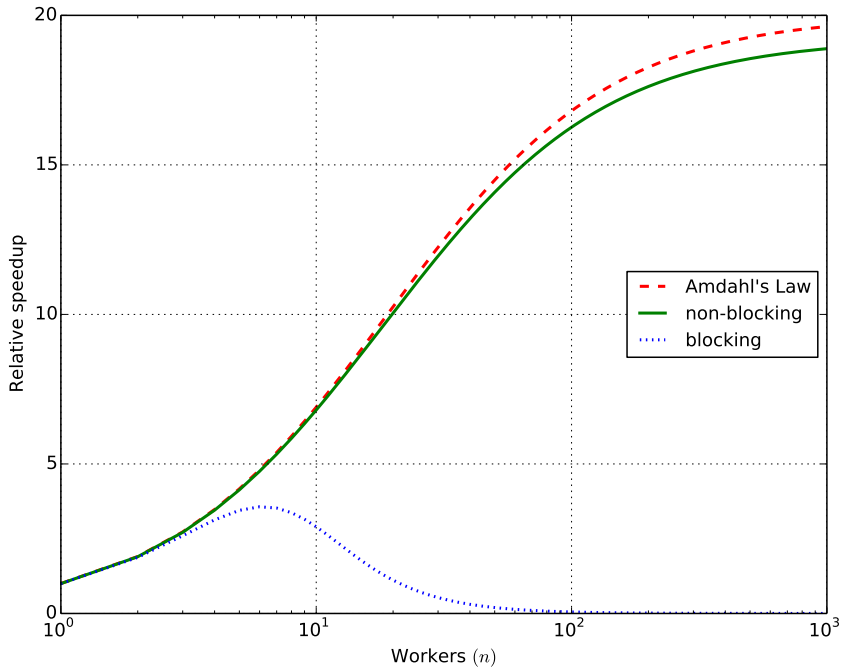


Figure 2.4: Performance models with overhead for the fixed size model with $s = 0.05$ and $\omega = 0.002$.

2.3 Summary

In this chapter, an overview of what parallelism means was covered in a qualitative and quantitative approach. Performance models for when the *problem size* is fixed and for when the *problem time* is fixed were derived. These models can be applied to the LHC context (which is characteristically embarrassingly parallel), where for the fixed size model the number of files used in an analysis is constant. For the fixed time model, the time to carry out the task is always constant. If the number of processors increases, so does the problem size. A basic model with communication was then developed for the fixed size model.

It was found that for the fixed size model the relative speedup is dependent on the fraction of code that has to be run sequentially (s). When s is large, efficiency drops off quickly making the addition of more workers pointless. For the fixed time model the relative speedup scales linearly with the number of processors. As the number of processors grows, efficiency declines slowly. The performance model with communication applies to a best and worst case scenario for the fixed size model: A network that only allows one message to pass at any time (blocking), or a network that can allow all messages to pass at once (non-blocking). A realistic network will lie between these two scenarios.

Chapter 3

ARM Processors

ARM [29] (originally Acorn RISC Machines, now Advanced RISC Machines) was initially founded in 1990 by Acorn Computers with supporting funding from Apple Computers (now Apple Inc) and VLSI Technology. The goal was to design a range of processors that consumed very little power but were still able to function at speeds comparable to processors supplied by their main competition, Intel and AMD. As the name suggests it incorporates RISC (Reduced Instruction Set Computing) architecture as opposed to CISC (Complex Instruction Set Computing) architecture. A traditional computer needs more hardware to perform more tasks at once to increase overall speed. This also consumes additional power and takes up a greater amount of space. The ARM processor was designed to perform only a few instructions at once, but at much greater speeds. This reduces the need for hardware such as transistors, which decreases the physical size of the system on chip (SoC). This also minimizes power consumption.

ARM has seen an enormous leap in popularity in recent years with their architecture being used in around 95% of the mobile market, including smartphones and tablets. It is also not uncommon to find these chips in printers, dedicated multimedia devices, and gaming consoles. More recently, embedded devices (essentially a stripped down computer) have become popular providing a cheap alternative to the traditional computer. ARM is now attempting to enter into the server market in-which their recent 64-bit architecture will have the biggest advantage. This poses a threat to companies such as Intel and AMD, and forces them to consider price/power/performance ratio in their future designs. An important note is that ARM does not manufacture its own processors but rather sells the rights to their intellectual property. It is then up to the companies that buy the rights to build and sell the hardware.

This chapter explores the ARM architecture. Focus will be on the specific ARM CPU's used in this thesis, namely the Cortex-A7, Cortex-A9 and Cortex-A15 in the form of their respective development boards.[1] A description of how the power consumption was measured will be presented and finally a note on the choice of operating systems will be explained.

3.1 Cortex-A series architecture

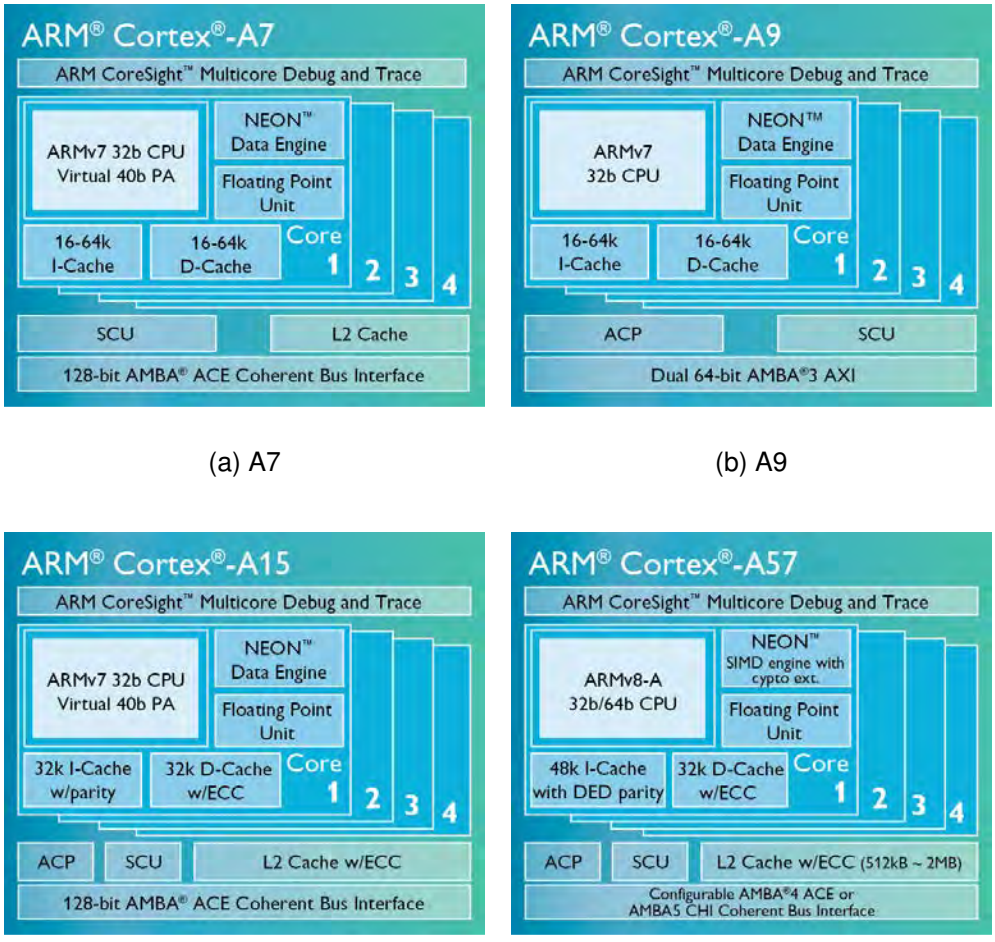
Cortex-A series processors (from hereon referred to as the A series) are used in a wide range of electronic products at present. Their low power consumption, high speed, and small size makes them the first choice for smartphone and tablet manufacturers around the world. Of the different series of ARM processors (R and M are the two other series specializing in areas from automotive to cameras and micro controllers), the A series are best suited for the complex tasks that are required in scientific computing and are capable of running various user friendly operating systems. The A7, A9 and A15 processor are all 32-bit architecture. The recent A57 is the first 64-bit architecture (armv8-A) and is found only in a select few hardware systems that are not widely available at this time.[2] For this reason no benchmarks are performed on the A57. However, the introduction of armv8-A has potentially large ramifications for the future of ARM in high performance computing.

Figure 3.1 shows a summary of the A series processors chosen for this thesis as well as the A57 for comparison. The A-series architecture has the ability for quad-core processors but it is up to the manufacturer to decide how many cores are viable for their designs. The A7 (Figure 3.1(a)) is ARM's most power efficient processor. It is the industry standard for entry level smartphones and tablets where battery lifetime is the foremost priority. The A9 (Figure 3.1(b)) starts to bridge the gap between power efficiency and higher performance ARM processors. The A15 and A57 (Figure 3.1(c) and Figure 3.1(d)) are defined as high performance processors with the A57 exploring the realms of 64-bit computing. The A9 is a slightly older chip and so has an older floating point unit (FPU) which provides hardware support for floating point operations. This means that it is only capable of version 3 of the vector floating point extensions (*vfpv3*) whereas the other processors are capable of version 4 (*vfpv4*). These optimizations are added in at the time of compilation. Another

[1] The Cortex-A8 was the first processor with armv7-A architecture and so is also the oldest. For this reason it will not be discussed.

[2] Furthermore, this hardware is so new that optimizations for compilers are still being written thus losing out on much of the expected gain.

exclusive ARM technology which all A-series are capable of is NEON™ technology. This is an advanced version of SIMD (single instruction multiple data) technology with the aim to implement low-level data parallelism when carrying out instructions. This optimization can be enabled at compilation time with the "*neon*" flag. The ARM CPU schematics may be well defined, but a large part of the performance comes down to the choice of peripherals, RAM and cache size and many other parameters, which are left up to the manufacturer. For this reason planning needs to go into choosing the right development board.



(a) A7 (b) A9

(c) A15 (d) A57

Figure 3.1: The schematics for the A series processors (images taken from [29]).

3.2 Development boards

Each development board in this study was chosen for a variety of factors. The foremost and most obvious question was what type of processor would be ideal. While the A15 is supposed to be the faster processor, would the supposedly increased power consumption still make it a better pick over the A7 when it comes to the performance/power ratio? Important considerations were the clock speed and amount of cache and RAM that the CPU has access to, and so a selection from the A7, A9 and A15 was therefore chosen to compare results.

The choice of development boards is shown in Table 3.1[3] with their associated pictures shown in Figure 3.2. The Nvidia Jetson TK1 makes use of the big.LITTLE™ ARM technology. This means that in addition to CPU frequency scaling, there is an extra energy efficient processor (such as an A7) that will become active when computationally low tasks are being carried out. A hard float operating system (OS) is the most common type among ARM development boards. It means that the operations are done using the hardware such as the FPU rather than delegating the task to the OS. L1/2/3 cache refers to the location relative to the CPU. L1 is the closest and fastest of the caches. L2 is slower but often larger. L3 is the slowest and potentially largest. Having only a small but fast cache isn't ideal as this results in more cache misses. A larger L2 can sometimes be better than a smaller L1. The development boards are compared to Hep306, a "typical" desktop computer.

An important note is that of the graphics processing unit (GPU) capabilities on the Jetson TK1. Offline HEP computing software is not designed to make use of GPUs, it would require an entire rewrite of the software. This is not feasible due to the complexity of the software and the amount of money already invested. This will become more clear in Chapter 5. Thus, for this thesis, the topic of GPU's is overlooked. However, it is of interest that this board has a Kepler GPU with 192 low powered CUDA cores claiming up to 326 *Gflops*. This is potentially a large increase in computational power. Future HEP computing or "big data" in general can take this into account when they design their software.

[3]The terms GiB, MiB and KiB refer to gibibyte, mebibyte and kibibyte, respectively. These are the more correct units of measurements when referring to memory and cache sizes in computers. 1 GiB=1.07374182 GB, 1 MiB=1.048576 MB and 1 KiB=1.02400 KB.

Table 3.1: The different setups with key features.

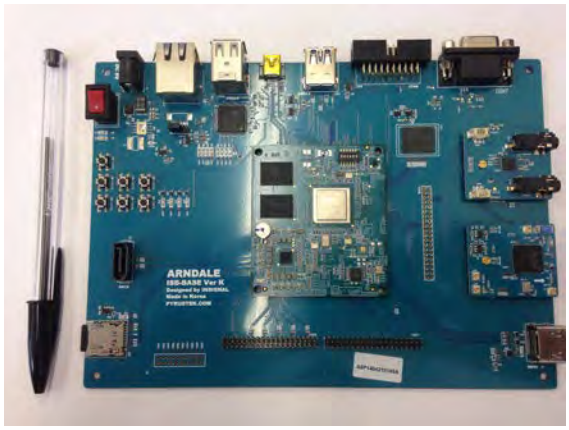
Setup	Processor	Cores	RAM	Cache	FPU	OS	Cost
Cubietruck	AllWinner A20, 1.2GHz	A7 dual core	2GiB DDR3	512 KiB L2	VFPv4	Archlinux, hard float	$89
Wandboard-Quad	Freescale i.MX6 Quad, 996MHz	A9 quad core	2GiB DDR3	32KiB L1, 1 MiB L2	VFPv3	Archlinux, hard float	$129
ArndaleBoard-K	Samsung Exynos 5250, 1.7GHz	A15 dual core	2GiB DDR3	32 KiB L1, 1MiB L2	VFPv4	Fedora 19, hard float	$220
Jetson TK1	NVIDIA Tegra-K1, 2.3GHz	A15 quad core-plus-one	2GiB DDR3	2 MiB L2	VFPv4	Fedora 21, hard float	$217
Hep306	Intel® Core i7-2600, 3.4GHz	quad core	16GiB DDR3	256KiB L1, 1MiB L2, 8MiB L3	-	Scientific Linux CERN 6	≈$600

3.3 Power measurements

The power consumption for the ARM boards was measured using a Fluke 289 Digital Multimeter [30]. The multimeter was placed between the wall socket and the power socket on the board, therefore this measured the development board as a whole which includes the CPU and extra peripherals. To obtain the power usage of just the CPU, the idle power was measured and subtracted from the measurement in question when performing benchmarks. The Intel® Power Gadget [31] was used to measure power consumption in Hep306. This software application measures the power consumption using only the energy counters in the processor. Thus it excludes all the peripherals. However, if the power usage characteristics and temperatures vary from those used in Intel's calibration, there will be errors between estimated and actual power usage. For this reason Hep306 serves more as an estimate than a benchmark.

(a) Cubietruck (A7)

(b) Wandboard-Quad (A9)



(c) ArndaleBoard-K (dual core A15)

(d) Nvidia Jetson TK1 (quad core A15)

Figure 3.2: The development boards used in this thesis.

3.4 A word on operating systems

The OS plays an important role in any type of computing, even more so in HPC/HTC as maximum performance is needed. It is mostly dependent on the version of the kernel and how well it is optimized for the hardware. RedHat Enterprise Linux (RHEL) [32] has been the favored OS for HEP computing where currently all servers at CERN run a rebuild of RHEL called Scientific Linux CERN (SLC) [33]. CentOS 7 [34] is recently being introduced and so SLC will soon become phased out. These OS's are very specific to scientific computing and not in high demand for your average user. Furthermore, there are differences in Linux flavours that some software compiled at CERN will exploit and

cause awkward failures when run on a derivative of Debian.

Functional Linux operating systems have progressed dramatically for the ARM architecture. However, choosing and installing an OS for an embedded device can be a complex and tedious task. The manufacturer's predefined kernel and OS are straightforward to install but they are always Debian based. Currently there are no properly functioning versions of RHEL or CentOS for ARM,[4] and so the choice was made to use Fedora on ARM (a RedHat derivative) when building/running ATLAS specific software. As will be shown later, this considerably and negatively impacted the performance in the STREAM benchmark.

3.5 Summary

This chapter focuses on the hardware and setups used in this thesis, namely the ARM processors. Background information is given on the ARM processors as well as an explanation on the architecture. While each series of processor used in this thesis (A7, A9, A15 quad and dual core) has a predefined schematic, it is also largely up to the manufacturer (who buys the rights to the ARM architecture) to decide on the final design of their SoC. Thus, there can be performance differences for the same processor but from different companies. The four development boards chosen for this research are the Cubietruck, Wandboard-Quad, ArndaleBoard-K and NVIDIA Jetson TK1. A discussion on how their power usage was measured as well as how the choice of operating system can affect performance was also presented.

[4] RHEL released "RedSleeve [35]" for ARM but it was built for armv5-A and so is outdated. CentOS is currently working on an ARM version but at the time of writing this is overdue.

Chapter 4

Benchmarks

The performance of a computer is a difficult characteristic to quantify. Some tasks require an enormous amount of CPU power whereas others require a higher memory bandwidth in order to push through data. It's also fairly simple to modify a benchmark to perform extraordinarily well on only one specific platform. While this looks good on paper, it then fails to perform as well in real-word applications.

There are two main classes of benchmarks; a "synthetic" benchmark and what we will call a "real-world" benchmark. A synthetic benchmark is an ideal case for a standard type of task such as reading data, performing algebraic tasks or any other manipulation. The goal is to obtain the best results as long as the modifications fall within specified guidelines from the developer. An example of a possibly allowed modification would be memory alignment; something that your average user will rarely do since the gain is often not worth the effort. While the results can be impressive these modifications are hardly ever done. However, they enable one to compare various setups.

A real-world benchmark applies to an actual problem for computers to analyze. These are more effective for the obvious reason that they are true results as well as providing a means to compare setups. The downside to real-world benchmarks is that specialized software has to be built on the system in question. This is not straightforward if the system in question has an architecture or design which was not considered when the software was initially built.

This chapter has a dual purpose in that it benchmarks the given ARM processors thus creating a "ranking system". It also explores the ability to compile HEP specific software on the ARM architecture, some of which has never been done before. It needs to be noted

that benchmarking various ARM processors is not a new idea. The Mont Blanc Project was one of the initial groups to see the potential of the ARM processor in large scale computing [36]. A group within the CMS experiment at CERN were among the first to consider ARM processors in HEP computing [37]. Other groups at CERN such as LHCb and the University of the Witwatersrand, South Africa have also benchmarked similar ARM CPU's [38, 39, 40]. Each group brings a particular viewpoint to the way benchmarks were performed as well as their intentions for the ARM CPU. It is encouraged to read up on these results.

Three benchmarks are purely synthetic (HPL, STREAM, PMBW) and are trivial to run on any machine. We then look at an E/p analysis benchmark which is real-world followed by PROOF which is a combination of a synthetic and real-world benchmark. While running the ROOT framework on ARM is not unique, the way in which it was done presents a first to the author's knowledge. This will be made clear in the coming sections.

4.1 The HPL Benchmark

The LINPACK benchmark [41] is historically one of the most common CPU intensive tests in high performance computing being used as early as the 1980's. The High-Performance LINPACK (HPL) benchmark is the parallel version of the LINPACK benchmark which is used to rank the world's TOP500 supercomputers [23]. The benchmark outputs a series of results that describe the performance of the computer when applied to solving a dense matrix problem of the form $Ax = b$, where each element is a uniform random number between -1 and 1. Ideally the increase in speed is scalable, i.e. four cores is four times quicker than one core, however, communication and latency between the cores hampers the performance and so speedup is never actually 100%. The suite makes use of 64-bit floating point arithmetic and portable routines from the Basic Linear Algebra Subroutines (BLAS) and routines from a message passing interface. In this case the Advanced Tuned Linear Algebra Software was implemented to generate an optimized BLAS library and Open MPI was used for intra node communication.[1]

The user can specify how much memory to commit to solving the largest problem that the machine is capable of solving. It calculates the floating point operations per second or *flops* of a system by splitting the large matrix into blocks that are then solved on different

[1] Open MPI is also capable of intra node communication, however, it treats the problem as if it were inter nodal communication.

cores or CPU's. This enables several blocks to be worked on in parallel. A list of block sizes (*NBs*) and matrix sizes (*Ns*) are specified at the beginning of the run (in ascending order). Then each matrix size is iterated over using the increasing block sizes. Thus, there is an overall increasing trend in transferred matrix size as shown in Listing 4.1. There is also the ability to specify the topology which is more relevant in a cluster than on a single computer. This is shown as the processing grid, $P \times Q$, where the ideal grid topology for four cores is set as 2×2.

An important note is that the same *hpl.dat* file was used for all the ARM boards (with the obvious changing of topology and filenames for respective boards). The manner in which each test is iterated by increasing block size and matrix size means that a spectrum of array sizes are tested. If the array size is too big then the benchmark exits and no information is lost.

Listing 4.1: A relevant portion of the *hpl.dat* file used in the HPL benchmark suite.

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
wandboard_vfpv3-d16_hf1.out output file name (if any)
8         device out (6=stdout,7=stderr,file)
15         # of problems sizes (N)
600 1600 2600 3600 4600 5600 6600 7600 8600 9600 10600 11600 12600 13600
   14600 Ns
15         # of NBs
10 20 30 40 50 60 70 80 90 100 120 140 160 180 200 NBs
0         PMAP process mapping (0=Row-,1=Column-major)
1         # of process grids (P x Q)
2      Ps
2      Qs
.
.
.
```

Figure 4.1 shows the power consumption with the respective *Gflops* and *Gflops/*watt. Power measurements were taken at a 7 second resolution. The grey area is an "envelope function" which splits the data into blocks along the time axis and then finds the average power value for that window. It gives a representation of the average power consumption which is indicated by the blue line. A trend we see for the A7 and A9 is that as the size of the matrix block that is passed to the separate cores increases, the average power consumption decreases. This is because slower computation means that less work in

the form of communication has to happen between the processors. The average power consumption remains fairly constant with the A15 dual core and quad core because of the faster clock speeds. Interestingly, the A15 dual core sees very large spikes in power consumption, however, they happen too rarely to have much of an impact on the average values for a given time-frame. The *Gflops/*watt is calculated by taking the average power consumption for the time window where there is a HPL measurement. We can see that the A7 does not have a fast enough processor. The A9 is fairly efficient provided the block size is relatively large and the *Gflops/*watt is comparable to the output of Hep306 (if not greater). The A15 quad core performs the best as one would expect (although, not much better than the dual core due to the extremely low power usage). The combination of a faster clock speed and a very power prudent development board performs quite well when it comes to *Gflops/*watt.

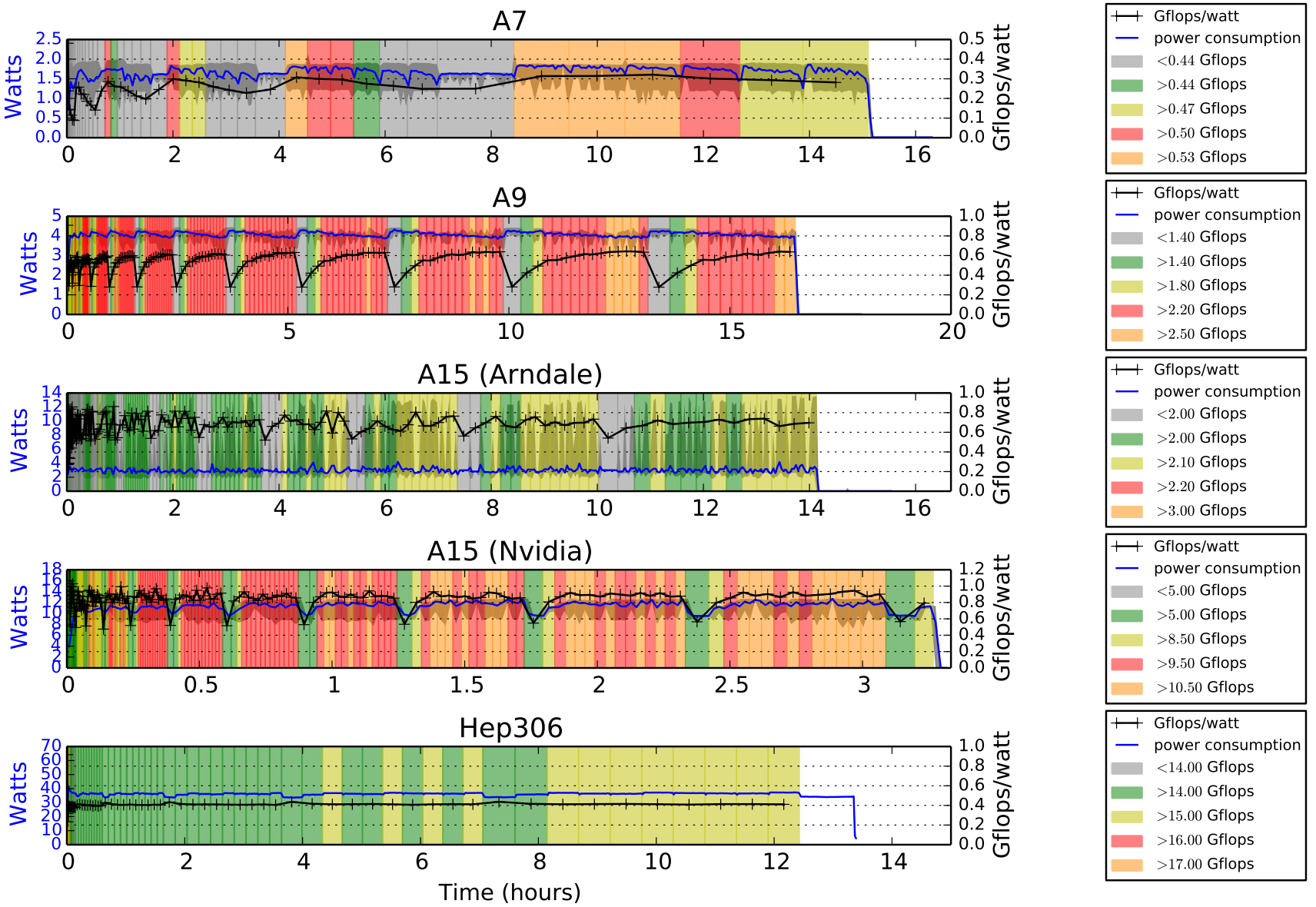


Figure 4.1: The relationship between matrix size, block size, *Gflops*, power consumption and *Gflops*/watt for the HPL benchmark.

4.2 STREAM

STREAM [42] is a simple synthetic benchmark used to measure peak sustainable bandwidth and kernel abilities for different computer architectures. There are four vector kernels that STREAM carries out and reports on:

- Copy: The most basic memory bandwidth test, $\vec{x} = \vec{y}$.
- Scale: A simple operation is introduced, $\vec{x} = c\vec{y}$.
- Sum: Another basic operation is introduced to allow multiple read/write operations, $\vec{x} = \vec{y} + \vec{z}$.
- Triad: Adds extra addition and multiplication operations along with overlapping read-/write, $\vec{x} = \vec{y} + c\vec{z}$.

The benchmark runs each vector kernel a specified number of times (default is 10) over a predefined array size where the best result is reported. Different compiler options were tested with the fastest option chosen and reported. These are *vfpv3-d16* for the A9 and *vfpv4-d16* for the A7, A15 dual core and A15 quad core.

Figure 4.2 shows the bandwidths for a single core on each of the development boards and Hep306. It is clear that the A15 outperforms the other ARM boards. Of interest is the poor performance of the A9. This is due to the fact that it has the slowest clock speed. An interesting outcome of this benchmark is the ability to compare operating systems and their respective kernels as is done with the A15 quad core. Nvidia and Ubuntu paired up to create a kernel that takes advantage of the specific drivers on the development board. The result is actually what one would expect given the board's specs. It's apparent that Fedora has yet to build an optimized kernel and so the memory bandwidth is drastically reduced. This is an issue that needs to be addressed since Fedora is of a closer architecture to CERN's computing platforms than Ubuntu. The question now arises as to how the memory bandwidth scales with more cores running in parallel. This will be addressed in the next section.

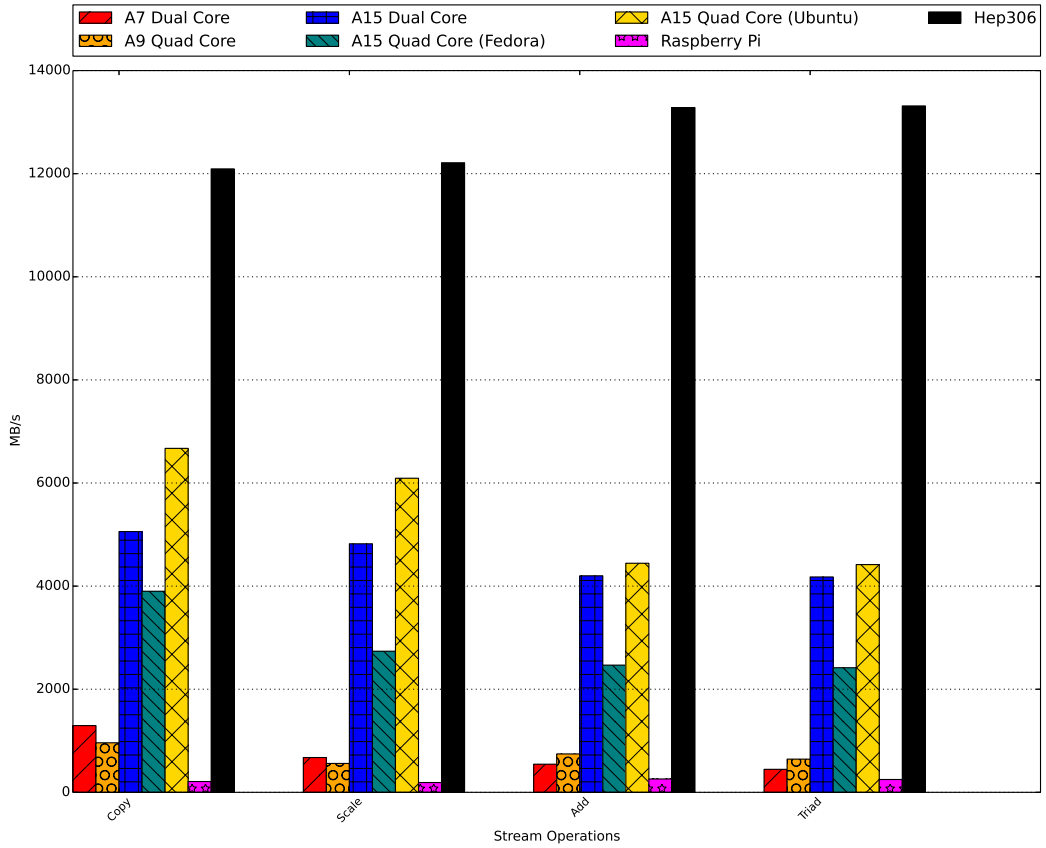


Figure 4.2: Stream results for the different ARM boards showing the memory bandwidth for a single core. For interest sake, the well known Raspberry Pi has also been included.

4.3 PMBW

The Parallel Memory Bandwidth Benchmark (PMBW) [43] is a suite that measures bandwidth capabilities of a multi-core computer. This is an important test because more cores result in the floating point performance increasing in a linear fashion. However, if the memory bandwidth is not capable of processing the data fast enough those processors will stall. Unlike floating point units the memory bandwidth does not scale with the number of cores running in parallel.

The code was developed in assembler language which means compile flags for the SoC become unimportant, thus there are no optimizations. It starts off by assigning a total array size to the highest power of two that still fits in the given RAM, i.e., $2^n < RAM$. For all the ARM boards the total RAM size is 2 GiB, thus $n = 30$ which equates to the

largest array size of 1GiB. The benchmark then steps through each assembler routine while increasing the array size as well as increasing the number of threads, (p). The array is split evenly between the threads. A small selection of the assembler routines have been chosen as follows:

- *ScanWrite/Read64PtrSimpleLoop* - Scanning operation while performing write and read tests. This routine transfers 64 bits (8 bytes) at every instruction. *Ptr* refers to pointer iteration[2] and *SimpleLoop* means only one iteration is carried out with each instruction.[3]
- *ScanWrite/Read32PtrSimpleLoop* - Similar to the above, except 32 bits are transferred at every instruction.
- *ScanWrite/Read32PtrMultiLoop* - Similar to the above except that *Multi* refers to the ARM specific multiple register transfer instruction. This reduces overhead from fetching instructions and so should be a faster memory access pattern.

Each of these routines has been plotted in Figure 4.3 with the routine name above each graph. They show the bandwidth in GiB/s for one thread as well as four threads, ($p = 4$). Threads between these values as well as threads higher than these values are omitted so that the graphs are easier to read. References to various cache sizes of interest are shown on the figure to give the reader an idea of where the different cache limits lie, (Table 3.1). The A15 quad core (labelled as QC on the diagram) outperforms the other boards in all areas. This is due to several factors; the clock speed is considerably higher and so is able to push through more instructions, the cache size is larger (even if it is L2) and so results in fewer cache misses, and it's a newer board, released in 2014. For single threads, the A15 dual core (labelled as DC on the diagram) performs better than the A9 and A7 as expected, due to the higher clock speed. Where $p = 4$ for small cache memory transfers the peak bandwidth for the A9 and A15 dual core are comparable, but as the array size increases and bandwidth starts to stabilize, the A9 doesn't perform as well as the A7 and A15 dual core due to the lower clock speed at 996MHz. The A15 dual core consistently experiences spikes when using four threads. One reason could be due to the dual core processor, where each core has to share resources between 2 threads. For reference, on a single thread Hep306 is able to reach a peak bandwidth of 121 GiB/s for

[2] A pointer is a variable that holds a memory location. The other option (not presented) is index iteration.
[3] The other option (not presented) is UnrollLoop in which more operations are carried out per loop but there is potentially more overhead.

a 256 bit message size while reading and 60GiB/s while writing. It reaches 298GiB/s for reading and 163GiB/s for writing when running 4 threads.

Figure 4.4 shows the speedup from one thread to four threads for the same benchmarks shown in the previous figure. The benchmark normalizes the time for each test so that it will take a minimum of one second per array size. For this reason equation 2.4 is used to calculate relative speedup. For *write* tests the maximum achievable speedup for the A15 quad core is approximately four. This drops off sharply as the L2 cache size limit is reached. It is then more beneficial to have a slower memory access pattern such as *SimpleLoop* rather than *MultiLoop* which causes bottlenecks when all threads are trying to access the cache simultaneously. The A9 and A15 dual core both have similar cache sizes but it's clear that in this case more cores is better until the array size is too large for the caches. Once bottlenecks start forming the clock speed is the determining factor. The *read* tests show much greater speedup since reading operations can be done more quickly. Reading data from the respective caches is clearly faster but fluctuates because of the hit/miss ratio. For this reason reading from RAM is more stable but also slower. These graphs shows that although speedup can and does occur for memory bandwidth applications running in parallel, they are not necessarily scalable.

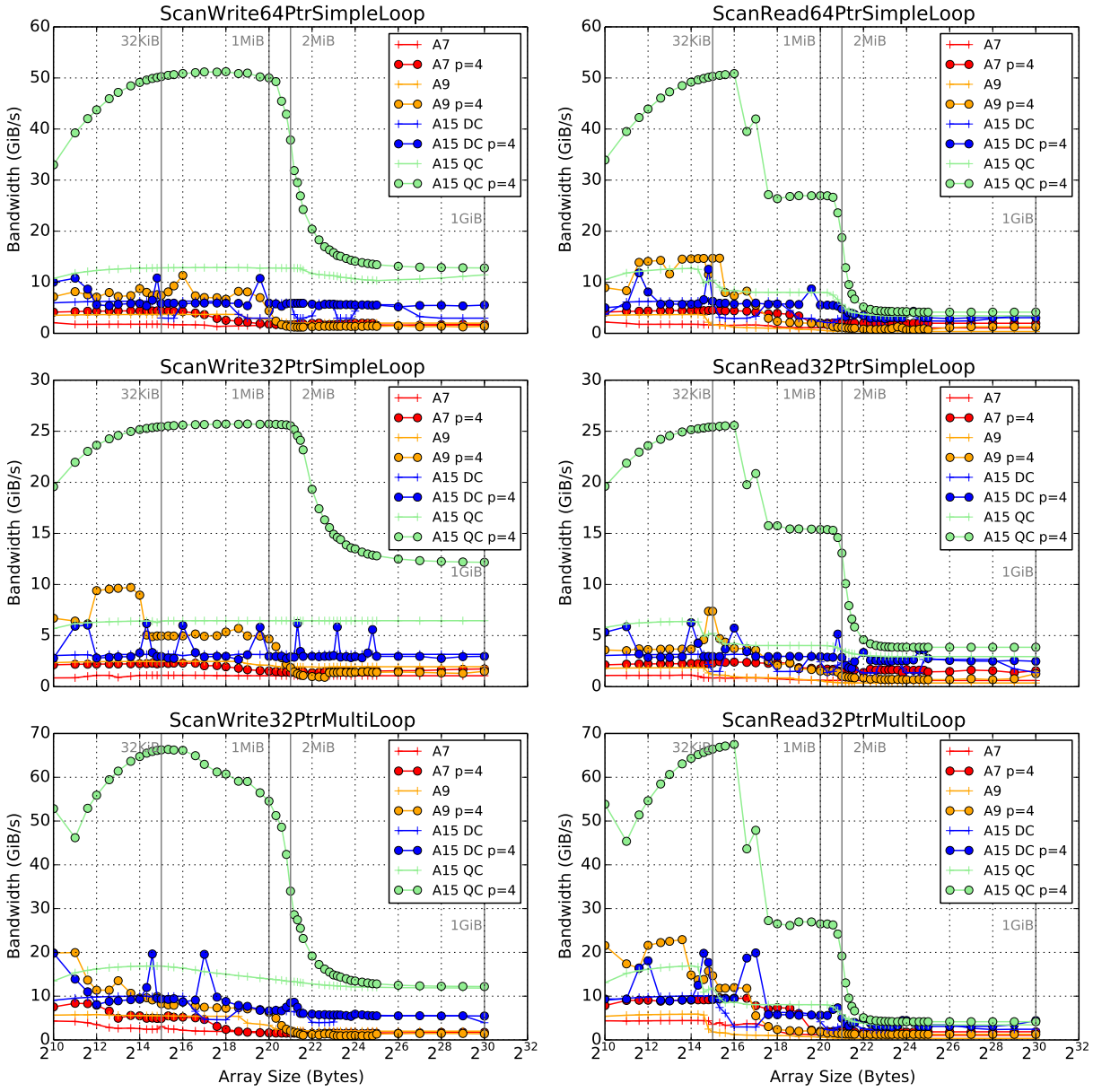


Figure 4.3: Memory Bandwidth given in GiB/s for selected assembly routines from the PMBW benchmark.

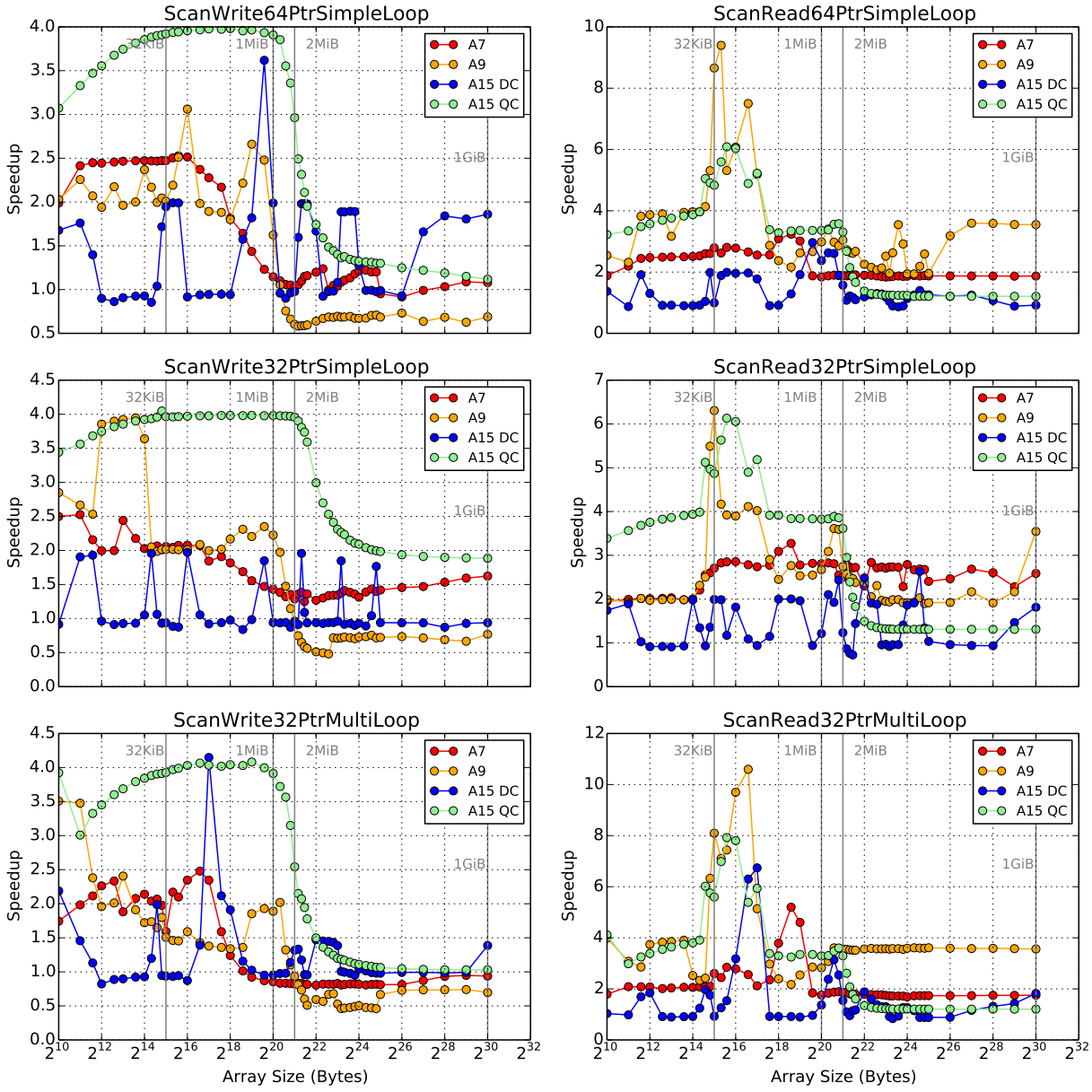


Figure 4.4: The speedup from one thread to four threads in the PMBW benchmark assembly routines.

4.4 E/p analysis for the TileCal

The TileCal was introduced in Chapter 1.4. It needs to be calibrated and monitored continuously, as is the case with any detector component. Calibrations are done in the form of charge injection systems, cesium systems, laser systems and the integrator system. More information on the calibration tests can be found at [44]. Once calibration has taken place performance monitoring occurs. One such test is to do an energy over momentum (E/p) analysis and compare the results to Monte Carlo data. Isolated hadrons are selected where their ionization energy (E) is measured in the TileCal. The hadron's momentum (p) is measure by the tracking detectors. The ratio of E/p is then compared to some independent variable such as the pseudorapidity (η).[4]

An E/p analysis was carried out on the A15 quad core (labelled as A15 QC on diagrams) with the code supplied by Carlos Sanchez from CERN. This real-world benchmark was important because it makes use of ATLAS specific code as well as demonstrates the use of the Configuration Management Tool (CMT) [45] on ARM. This tool is central to the ATLAS software framework and is described in more detail in Chapter 5 and Appendix A. The E/p analysis was built to run within the LXPLUS[5] service offered at CERN. The code was modified to compile locally, checking out the packages needed from the respective subversion (SVN) [46] repositories hosted at CERN.

This is an I/O and CPU intensive analysis with the goal of comparing the time taken for a fixed number of events to be analyzed when running on an increasing number of cores. This was done on the A15 quad core as well as Hep306. From this we can calculate relative and absolute speedup. A total of 1 dataset (approximately 250 MB), 59 files and 27489 events was used. As mentioned in Chapter 2, there are different approaches one can take when parallelizing this type of task. It was decided that for a HEP analysis the ideal approach is to assign N files to a single core. Thus, a script was written to assign N/n files to a single thread and the number of threads was then varied. By default, if there are four cores and four threads, each thread will be assigned to a core. Should the number of threads be greater than the number of cores, resources from a single core then have to be shared. Thus, one can start many threads, but at some point there will be a decrease in the performance. It should be noted that it was deemed necessary to carry out this benchmark on the quad core A15 running Fedora 21. As seen from the Stream

[4] Pseudorapidity is a spatial coordinate in particle physics. It is similar to θ in that it shows an angle from the beam axis, but it is invariant when boosting frames.

[5] This is an interactive service provided by CERN. Once logged in, it provides access to the ATLAS software framework, Athena where one can carry out many tasks including a full data analysis.

benchmark this choice of OS does hamper performance due to the kernel. However, this OS is the closest to that used by CERN, which is important when compiling specific software.

The results for the total time taken, relative speedup and the cost of the run are shown in Figure 4.5. For the first graph it's no surprise that the ARM processor is significantly slower than Hep306.

The middle graph shows relative speedup with a fit using the fixed size model with an embarrassingly parallel, blocking network (FSEB) shown in Equation 2.13. This produced a better fit than the non-blocking scenario and thus, was chosen. The serial fraction parameter was held fixed at $s = 0.01$, which produces an overhead of $\omega = (0.6723 \pm 0.0419)s$ and $\omega = (0.0641 \pm 0.0049)s$ for the A15 and Hep306, respectively.

The bottom graph, showing the cost of the run, has been calculated by taking into consideration the time of each run (t) on n cores and the average power consumption for that particular run. Thus,

$$Cost\ of\ run = Power[kW] \times t[h] \times \frac{116[c]}{1[kWh]}. \tag{4.1}$$

The cost of the run is quoted in the South African currency (ZAR) with an electricity rate of 116 c/kWh.[6] It is here that we see what translates to a distinct advantage towards the ARM processor in terms of *Gflops*/watt, and that it is cheaper running the analysis on the quad core A15. The ARM board is consistently cheaper by at least a factor of two.

Figure 4.6 shows $1/S_A$, (Equation 2.8) portrayed in a contour plot. It's clear and sensible that the "largest" absolute speedup occurs when the analysis is run on four threads for the ARM processor and one thread for Hep306. As expected, at no point will $1/S_A > 1$.

[6]Depending on the exchange rate, one ZAR is approximately equal to 0.080 CHF and 0.083 USD (28/03/2014).

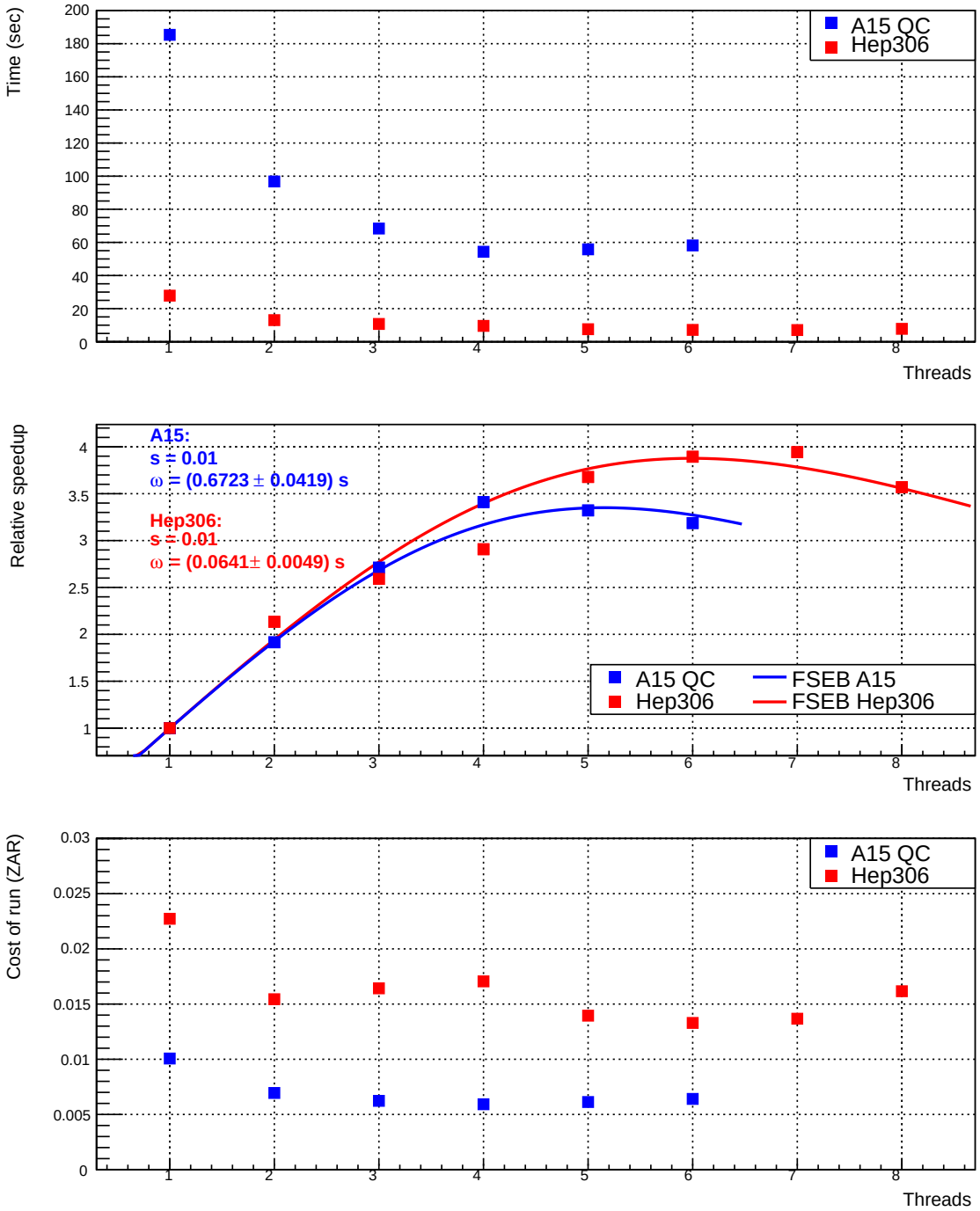


Figure 4.5: Top: The time taken for each E/p analysis to complete when performed on the quad core A15 and Hep306 with an increasing number of threads. Middle: Relative speedup with a fit demonstrating the fixed size model with an embarrassingly parallel, blocking network (FSEB). Bottom: The energy cost of each run normalized to ZAR.

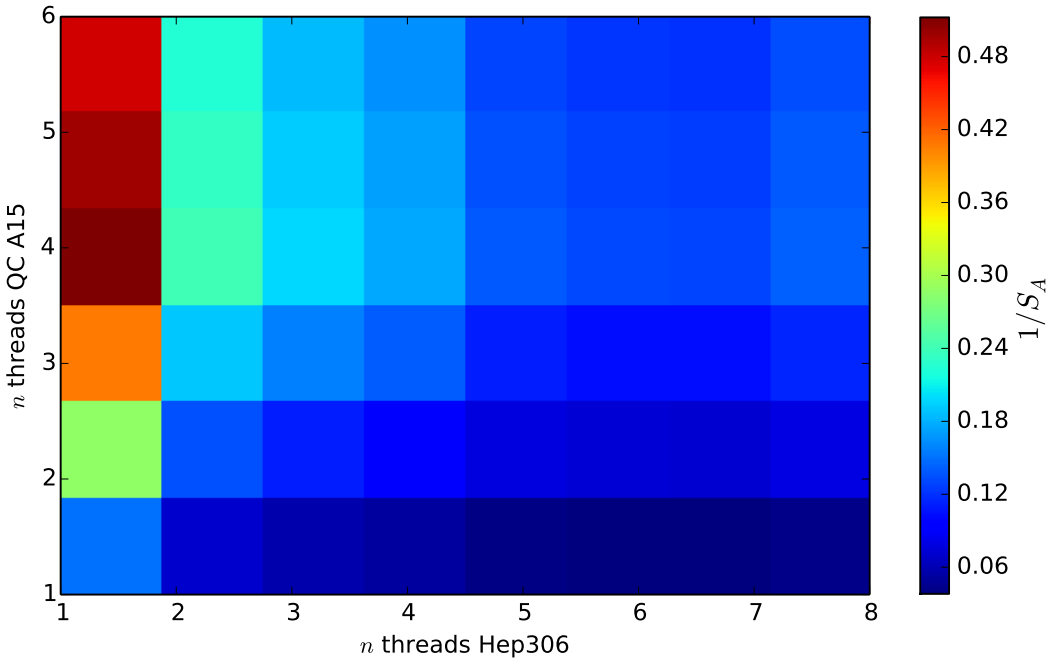


Figure 4.6: Results for the E/p analysis showing $\frac{1}{S_A}$.

4.5 The PROOF Benchmark Suite

In collaboration with the University of the Witwatersrand (Wits) physics department in Johannesburg, South Africa we built a dedicated PROOF cluster on ARM boards. To the author's knowledge this is the first of its type. It was comprised of seven quad core A9 boards (Wandboards) sharing a NFS directory hosted by a traditional server as shown in Figure 4.7. The daemon *xproofd* was needed to be initialized on each node so that incoming ROOT connections could be accepted. In an ideal situation each board would have it's own external storage device and file sharing would occur through XRootD [47], however as a proof of concept cluster this was adequate and effective.

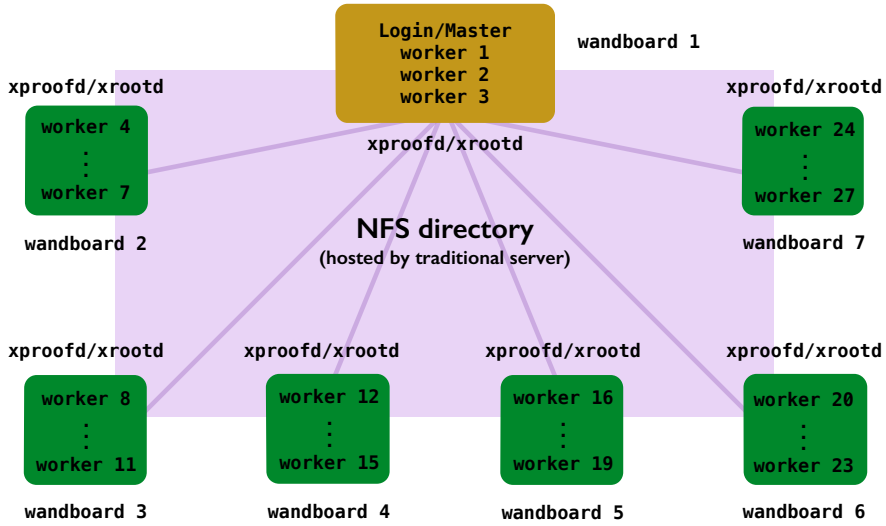


Figure 4.7: The setup of the dedicated PROOF cluster at Wits.

The PROOF benchmark suite [48] was used to determine performance. It is a further add-on designed to test the topology and scalability of clusters by running cycle- (CPU) and data-driven (I/O) processes. For these benchmarks ROOT was compiled with different ARM specific compile flags, specifically *vfpv3-d16* (where *d16* refers to 16 double-precision registers), *neon* and *default*. The average for each test was plotted. The two benchmark tests are as follows:

- The CPU benchmark: The default TSelHist selector was used with the output shown in Figure 4.8. The unit of measurement is the number of events per unit time. We can

see that the cluster scales linearly for CPU-intensive tasks. This is to be expected as the only significant constraint is the overhead produced when the master has to pool the results and create the histograms. This is evident in the right-hand graph showing the normalized events per unit time. Once the master core has to start co-ordinating jobs there is a slight decrease in speed, but this is quickly rectified.

- The I/O benchmark: The default TSelEvent selector was used with the output shown in Figure 4.9. This benchmark shows the bottleneck that the NFS server creates. Scalability saturates at around 12-13 workers which equates to reading events at a total of around 23 MB/sec. Its clear that a shared NFS directory reduces the advantage that one might gain from a PROOF cluster as can be seen from the normalized I/O graph.

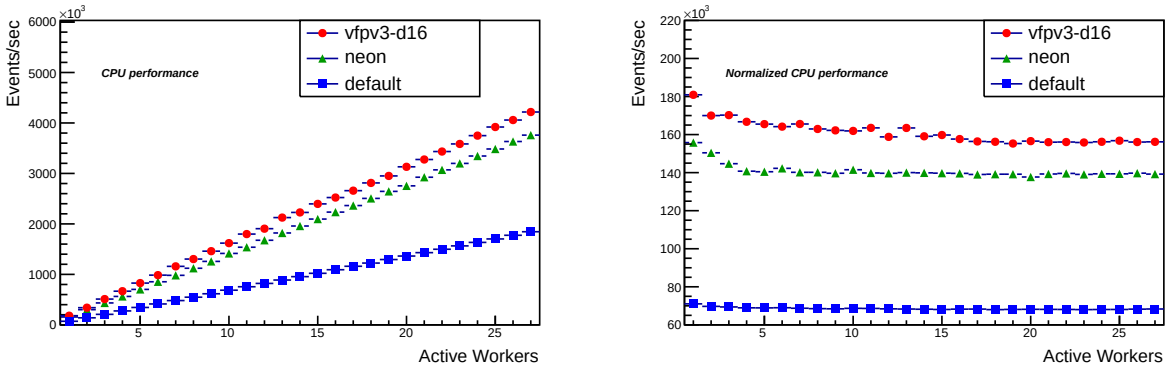


Figure 4.8: The PROOF CPU benchmark for the ARM cluster.

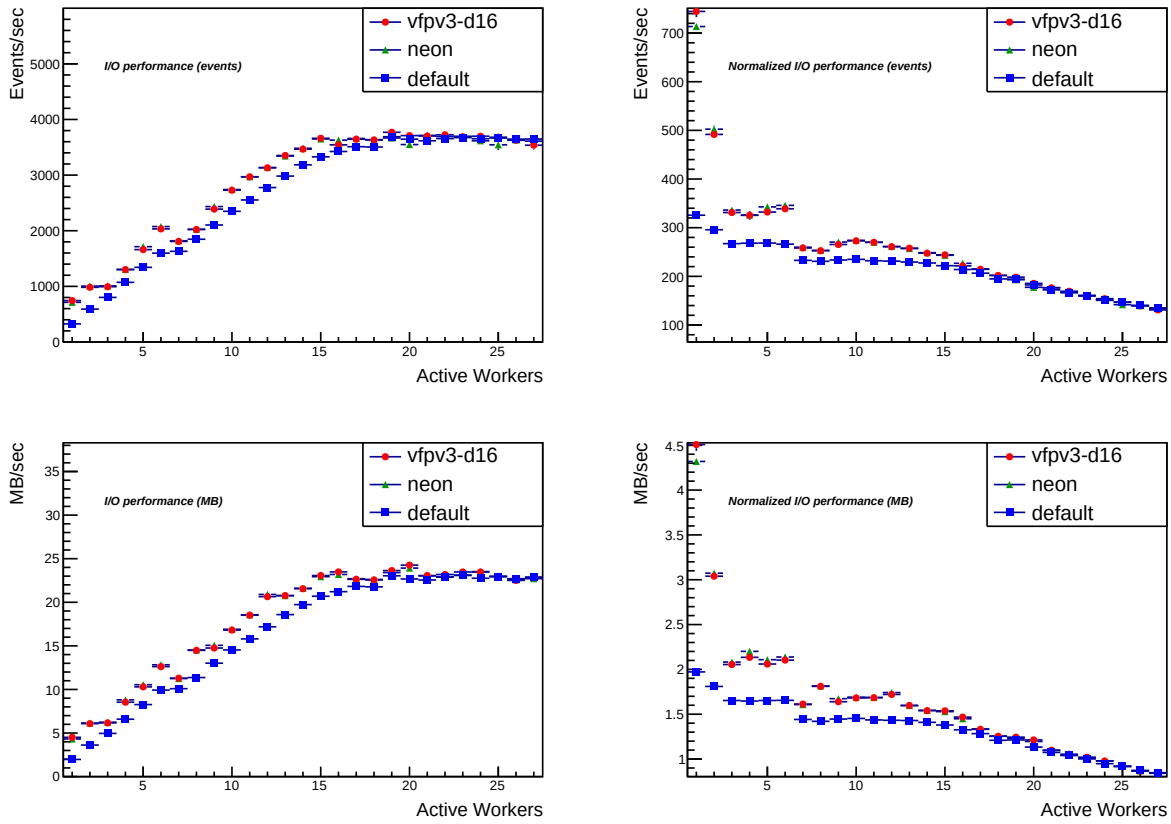


Figure 4.9: The PROOF I/O benchmark for the ARM cluster.

4.6 Summary

This chapter presents the results of various benchmark tests performed on the various ARM processors. The HPL benchmark proves that the A9 and A15 processors produce a better performance per watt ratio than Hep306. The STREAM and PMBW benchmarks respectively show that different OS's produce different memory bandwidth results, and that memory bandwidth does not scale with parallel computing. The E/p analysis was then performed comparing the performance of the quad core A15 to that of Hep306. This showed a practical use of CMT as well as a full ROOT analysis on ARM. The fixed size model with communication produced a reasonable fit when fitted to relative speedup. It shows that is cheaper to run the same benchmark on the ARM boards, but computation time is much slower. In order to achieve performance similar to that of Hep306, more cores are needed which makes energy savings negligible. Finally, the first dedicated PROOF cluster was built using A9 quad core processors. It showed good scalability but I/O was hampered by the NFS directory. It was however, a successful proof of concept. A consistent theme is that the A15 quad core performs the best of all the ARM processors in terms of performance and performance per watt.

Chapter 5

ATLAS Software on ARM

With the performance of ARM processors improving, it is necessary to port the entire ATLAS software stack to the ARM architecture, something that has never been done before. The goal is to run all the ATLAS software on ARM processors. This will enable server farms comprised entirely of ARM processors to function the same way as the traditional Intel and AMD server farms.

The software stack used by ATLAS, Athena (named after the Greek goddess of wisdom) is complex and dynamic. Thus, it is more practical to separate the different packages into individual projects and then build from ground up. Fundamental projects known as LHC Computing Grid (LCG) software, GAUDI [49] and ATLAS specific code comprise an Athena release where the latter, by itself, consists of over 6 million lines of code. Roughly 4.5 million lines are written in C++, 1.4 million in python, while other languages make up the rest. It is estimated that the cost of development for release 17.2.3 was $231 million with an accumulated 1709 years of development put in by developers.[1] Athena versions are compiled using the NIghtly COntrol System (NICOS) [50] which facilitates the building of the individual packages and projects for this large framework. Complete versions are compiled every night for the x86_64-slc6-gcc48 platform (i686-slc6-gcc47 and x86_64-slc6-gcc47 are also compiled for older versions of Athena) with standard tests being carried out once compiled. Fixes are performed the next day once bug reports have been generated. NICOS compiles the releases using CMT. It should be noted that there is a migration to using the faster more flexible tool, CMake [51]. This has many advantages. The first is user friendliness. CMake is used by more frameworks (not limited to HEP computing) than CMT and so is more user friendly with a larger pool of documentation. It

[1] Statistics from communication with Rolf Seuster.

is a cross compiler and so opens up new avenues of being able to compile on alternative systems to that of the host. This is very important because it means that an Athena compilation being controlled by NICOS has the ability to compile releases for ARM while not requiring any of the servers to have ARM CPU's. Finally, it is able to exploit parallelism by assigning different package builds to different threads and so speeds up the net compile time by significant factors depending on how many cores and memory are available.

Figure 5.1 shows the software structure for Athena release 20.1.0 (read from bottom to top). This is the release chosen to be ported to ARM, where the non-shaded projects were neglected in the compilation. These focus more on online computing. The rest of the projects were compiled knowing that certain packages would fail due to the dependencies on tdaqCommon and dqmCommon. Future compilations on ARM will include these projects.

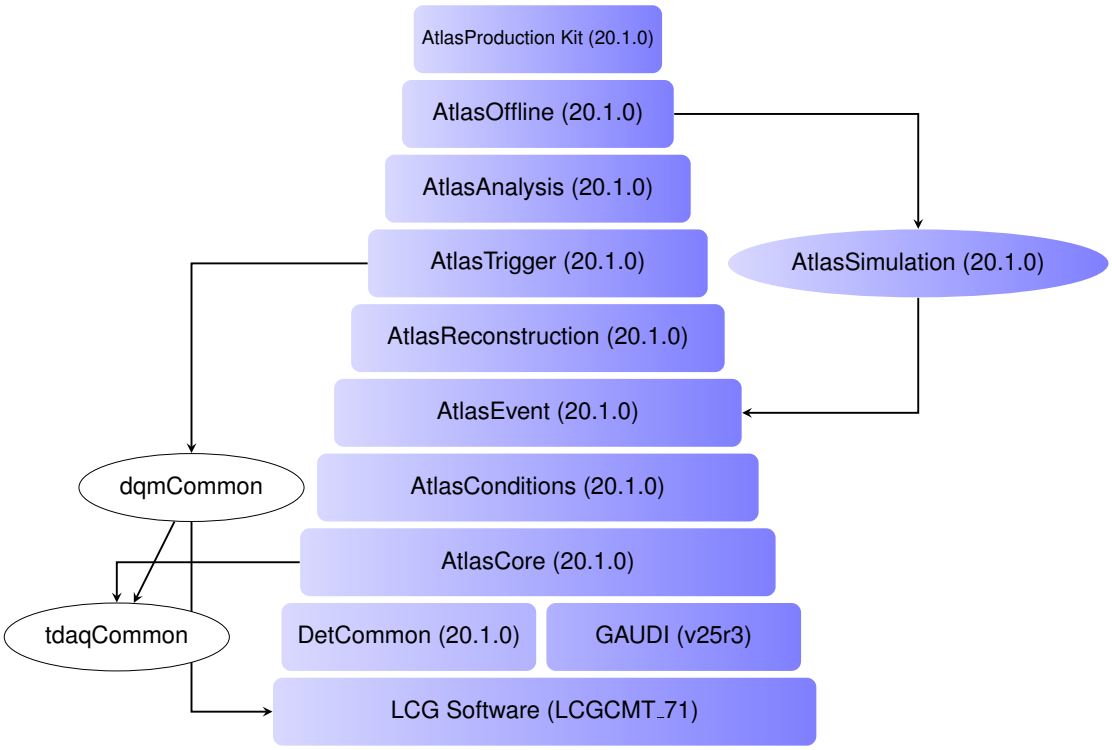


Figure 5.1: The software stack for an Athena release.

This chapter focuses on compiling the different stages of an Athena release starting from LCG software and GAUDI which were compiled natively using CMake. Then a newly written and portable framework, Athena Nightly on ARM (ANA) [52] will be presented which facilitates the compiling of the rest of the software stack on ARM using CMT. All software

compilations were done on the quad core A15 running Fedora 21. Understanding CMT is crucial to ANA, therefore it is presented in Appendix A. CMake is a considerably easier tool to use with more documentation available. Thus, it will not be presented in-depth.

5.1 LCG software

At the base of the pyramid we have the LCG software, crucial to all LHC frameworks and so independent of experiment. These are packages that can be downloaded and compiled fairly easily, however it is a very time consuming process as there are approximately 200 packages each with individual configuration flags. Furthermore, some packages have dependencies. Two examples of this are shown in Figure 5.2 and Figure 5.3 for ROOT and Python, respectively. These diagrams (created with DAGitty [53]) show examples of a small part of the complexity of the LCG software. They neglect to show the smaller libraries and executables that are needed, of which there are many. In the best case scenario a package that is missing a dependency will fail to compile. One can address the problem immediately and continue with the compilation. In the worst case, everything compiles with obscure errors showing up in awkward places at later stages. It is a time consuming process to recompile the needed packages.

The main goal of compiling LCG software using CMake is to create a reliable, automized and parallelized process of building the many packages that form the base of the pyramid. An example of a function that gets called to download, compile and install a package (with ARM flags added in) is shown in Listing 5.1. Each package has a similar entry and is parsed when the CMake build command is called. All the packages are installed into what we will call the *LCG_install* directory.

5.1.1 Major changes for ARM

Most of the LCG packages can be built without too many problems. However, there are some caveats:

- None of the external grid packages can be built. The reason for this is that they are specifically built for the traditional x86 computer architecture. Some of the packages can be built regardless, but it was decided to ignore them for the initial versions of

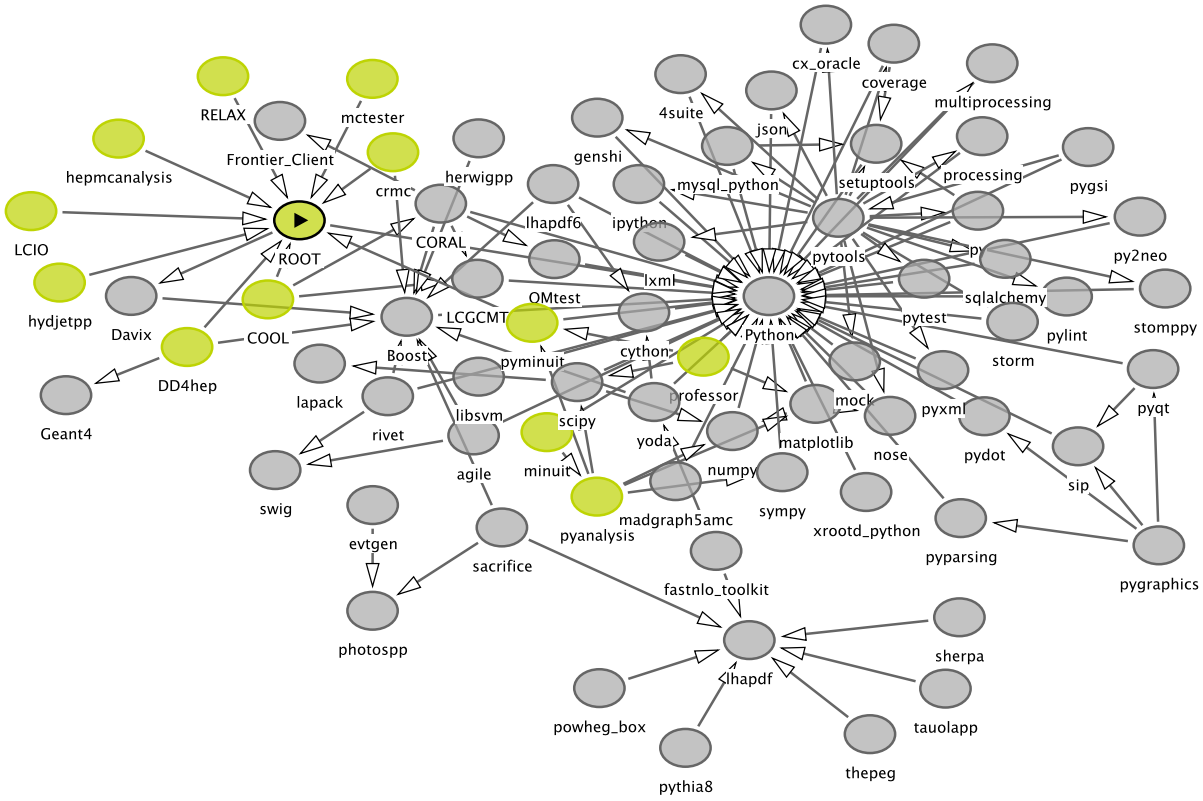


Figure 5.2: Directed acyclic graph showing packages dependent on ROOT.

Listing 5.1: An example of the CMake code for the LCG software with newly added “armv7l” flag option.

```
#---vdt---------------------------------------------------------------------
if(LCG_TARGET MATCHES "armv7l")
  set(vdt_extra_config_opts -D NEON=1)
endif()
LCGPackage_Add(
  vdt
  URL http://service-spi.web.cern.ch/service-spi/external/tarFiles
     /vdt-${vdt_native_version}.tar.gz
      CMAKE_ARGS -DCMAKE_INSTALL_PREFIX=<INSTALL_DIR> vdt_extra_config_opts
  BUILD_IN_SOURCE 1
)
```

Athena on ARM. These packages provide the means to connect a server farm to the LHC Grid. An ARM farm does not have to be connected to the grid to function, in fact, most servers/server farms (used for HEP analysis) at universities around the world are not connected to the grid.

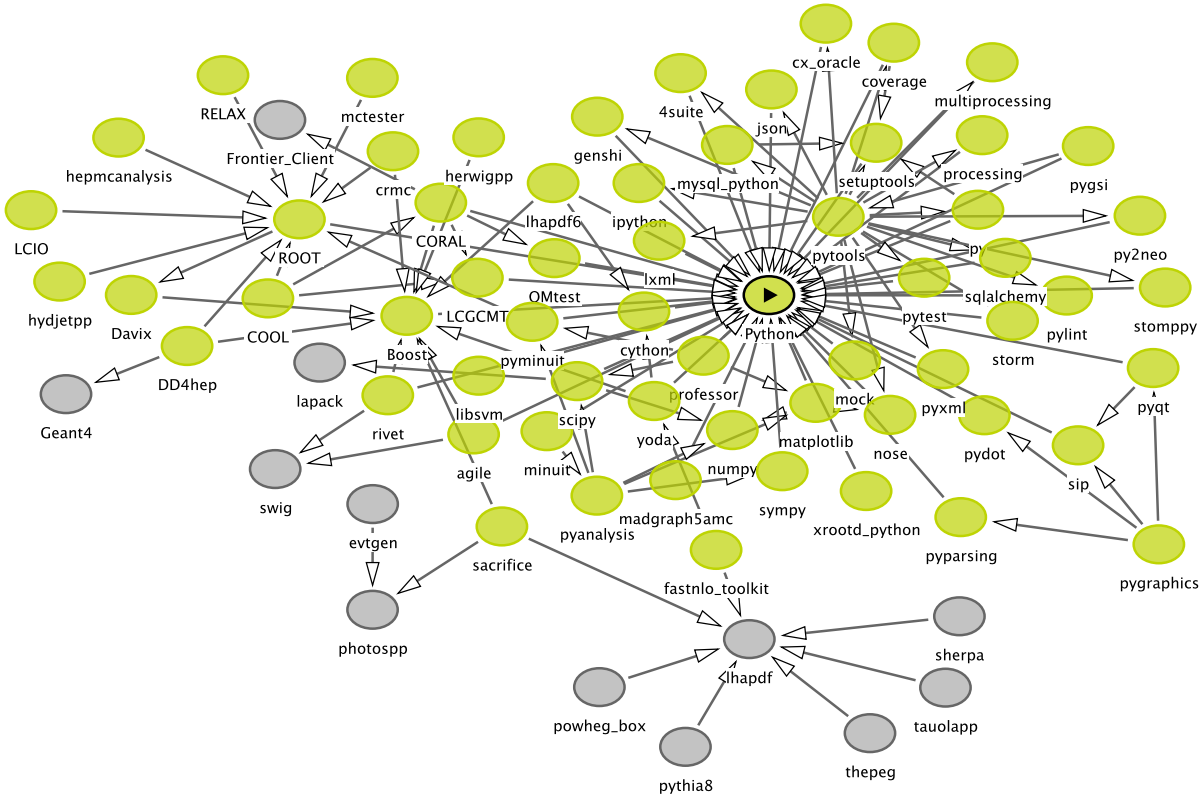


Figure 5.3: Directed acyclic graph showing packages dependent on Python.

- Many of the packages' source files are located on a server at CERN compiled for certain architectures, none of which are ARM. These URLs need to be redirected to alternative sites where general tar-balls are located.

- A significant setback is that Oracle DB does not have ARM 32-bit source code and doesn't plan to ever provide it. While they might provide it for the 64-bit architecture in the future this leaves a gap in the 32-bit Athena release with consequences that have yet to be understood. For this reason, the packages COOL and CORAL have been built as separate CMT projects rather than part of the LCG software. This is due to CORAL depending on Oracle libraries, and COOL depending on CORAL libraries. Building these separately provides a higher level of control. The code for each can now be modified to include or exclude the use of Oracle DB.

- An important package that requires editing is the LCGCMT package. CMake builds and installs the libraries and binaries of the LCG software but to proceed further one still needs the routines and patterns in CMT syntax applied to each installed

Listing 5.2: Tags used in LCGCMT project propagate throughout the entire release. This shows an example of default tags with the tag for ARM added in.

```
tag i686-slc5-gcc45-opt target-i686 target-slc5 target-gcc45 target-opt
tag x86_64-slc6-gcc48-opt target-x86_64 target-slc6 target-gcc48 target-opt
tag armv7l-fc21-gcc49-opt target-armv7l target-fc21 target-gcc49 target-opt
```

package. Thus, the specific LCGCMT project (in our case LCGCMT_71) needs manual editing. For example, in *LCG_install/LCGCMT_71/LCG_Platforms*, targets are used to steer CMT. Tags need to be created so that the ARM architecture is now defined. Listing 5.2 shows the new ARM tag defined for the setup we used to compile all the software. This at a later stage will be defined as Linux. Architecture specific compile options can now be added into the code.

Once the packages have been installed and the appropriate tags in LCGCMT project have been defined, one can move onto the compilation of GAUDI.

5.2 GAUDI

GAUDI is an object orientated framework used to facilitate common event data processing applications. The foundation libraries are compiled using packages from the LCG software which then contribute to from GAUDI's framework toolkit. This enables algorithms that pertain to online and offline data processing such as the high level trigger, reconstruction, event simulations and event visualizations. It is an external project that was originally developed for the LHCb experiment but was designed to be customizable and applicable to a large number of scenarios. Thus, it has been adopted by the ATLAS experiment and various other experiments that are not part of CERN. GAUDI is absolutely crucial for the ATLAS code to compile.

This project was compiled on the ARM architecture using CMake. For the most part, everything compiled as expected, however, there are two points worth noting:

- The application Intel® VTune™ is not ARM compatible and so this module was disabled.
- Earlier versions of GAUDI have not adapted their *instrset_detect* code to account for the ARM architecture. This code gathers information on the CPU which isn't crucial

for a functioning GAUDI release. Should an older version of GAUDI be required (as was the case here) a simple replacement of a new *instrset_detect.cpp* file with the old will fix the errors. This new code essentially says that if this is being compiled on ARM, skip this functionality.

5.3 ANA

ANA is a framework that strives to simplify and automate the process of compiling the ATLAS specific code. The objective is to be able to repeatedly and consistently compile versions of Athena for the ARM architecture with minimal input from the user. It is important to note that only the ATLAS code compiled using CMT has been included into ANA version 1.0. This includes the projects DetCommon, AtlasCore, AtlasConditions, AtlasEvent, AtlasReconstruction, AtlasTrigger, AtlasAnalysis, AtlasSimulation and AtlasOffline. It means that the LCG software and GAUDI need to be compiled beforehand and still require essential manual tweaks to accommodate the ARM architecture. This was described in the previous sections. If ANA is implemented correctly all that is required as an initial setup is to provide the environment variables such as the paths of the external projects. Once this has been done, sourcing the build script initializes structures and starts the compilation until a failure occurs due to the local compilation on ARM architecture. The programmer then writes a patch and stores it in the *patches* directory where it will be picked up by ANA. This will be described more in-depth in the following sections.

5.3.1 Implementing ANA

The ATLAS specific code all follows the same structure when compiling with CMT. For each package in a release, steering is supplied by the *requirements* file. Initially, all macros and environments are tuned for compilations using NICOS on x86 architecture. Thus, compile options and environments need to be changed. In these early stages of development it's not feasible to go through the entire ATLAS code looking for issues that may or may not appear. The approach taken with ANA is to compile the packages while taking note of what fails. The developer then writes a patch that can then be applied or reversed. This way, records are kept of what changes are made and in the case of an incorrect diagnoses this can be easily corrected.

The structure of ANA is shown in Figure 5.4 with an explanation of the more important files/directories shown on the figure.

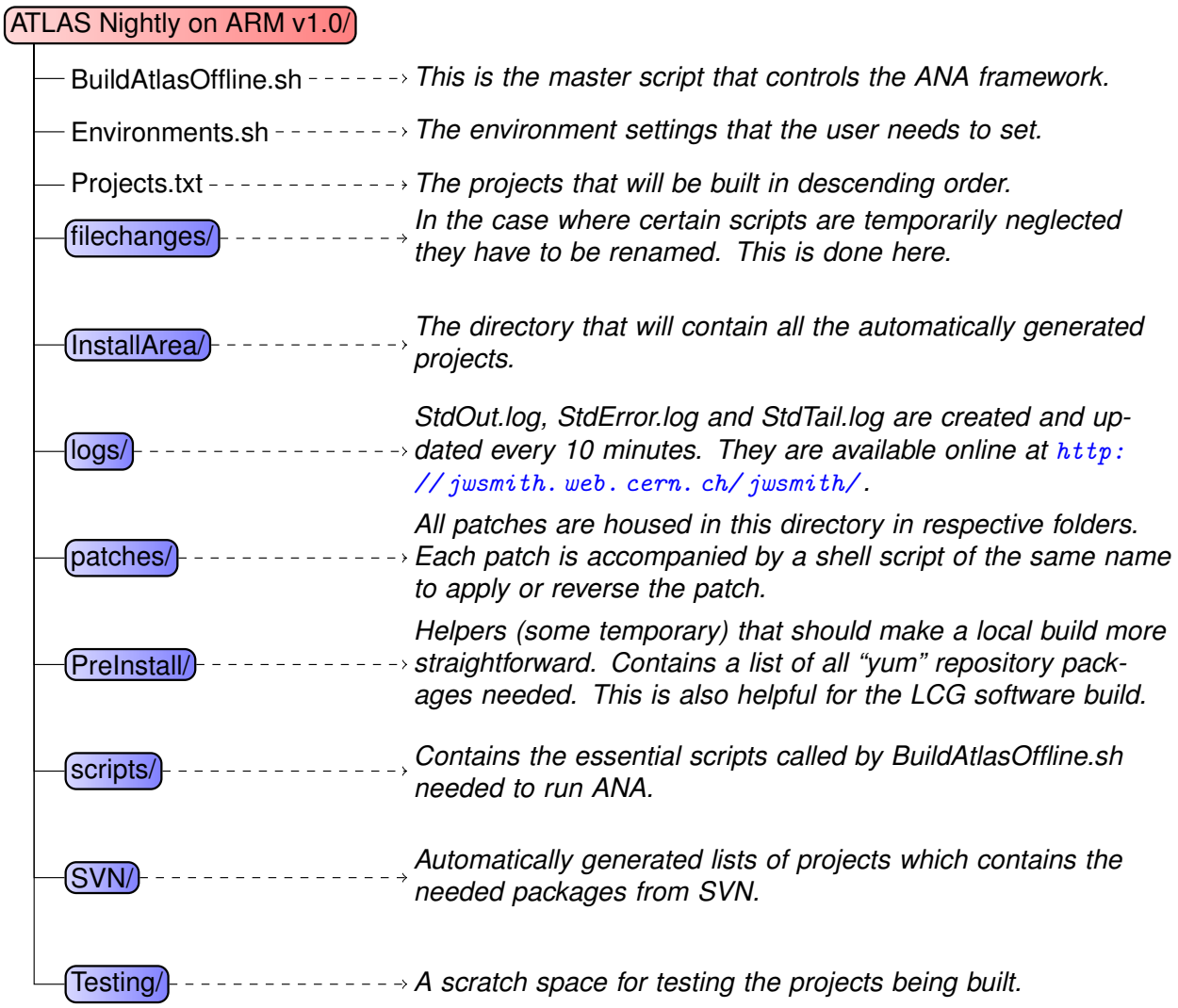


Figure 5.4: The structure and purpose of the important files and folders in the ANA framework.

To run ANA one needs the source code located at `https://github.com/jwsmithers/AtlasOfflineBuild-framework`. Once this has been downloaded all that is required is to set the environment variables in *Environments.sh* and then do

```
>> source BuildAtlasOffline.sh.
```

When this command is run:

- It creates the directory structure according to CMT rules.
- It creates the *project.cmt* file in each project directory.

- It creates and populates the *projectRelease requirements* files in the respective folders.
- It gives the option to pull the required SVN packages based on the *VERSION* variable set in *Environment.sh*. This is a lengthily process. If it hasn't done this before it will do so, otherwise it defaults to "no".
- It then provides the option to either apply all the patches in the *patches* directory or skip them. This defaults to "no". Patches can also be applied or reversed individually from their respective folders.
- It then gives the option to chose the project to compile (or by default, build all of them). This can also be set in the *Project.txt* file which is from where the default option reads.

Figure 5.5 shows the contents of the *InstallArea* folder as well as inside a project folder. Different release versions can be specified at anytime and built accordingly without the overlap of environment variables. However, while some patches will work for multiple releases others may be release dependent. For this reason patches are also sorted by versions.

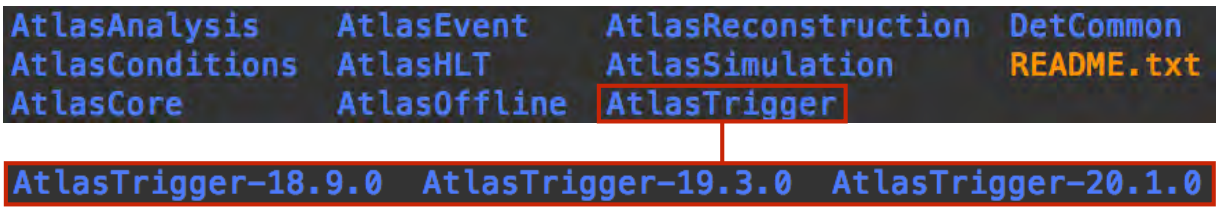


Figure 5.5: Inside the *InstallArea* folder and a project for ANA.

5.3.2 ANA patches

Most of the patches add the new ARM related tags into the code. They also fix path related errors due to the software being optimized for tools found in AFS and compiled for x86 architecture. There are still some oddities that need to be mentioned:

- Syntax errors: A few syntax errors were found such as missing "=" or "::" signs.

- Linking errors: Some libraries are linking in the wrong order and thus fail the first time being compiled. The temporary solution is to do a “make” twice.
- *Genreflex* compiler flag issues: The compile flag *-fPIC* doesn’t get removed when the *lcgdict* macro is called. Thus, it sees it as passing an option and so fails. The solution is to add a private macro which removes this option every time *lcgdict* is called. It is interesting because the options *-fsigned-char* and *-fsigned-bitfields*[2] are specified in the same place as *-fPIC*, yet these don’t seem to cause problems. This could stem from a tag being defined incorrectly.
- Virtual memory use: Occasionally a package would fail to compile because it exhausts the virtual memory address space. A temporary fix is to just recompile the package.

These highlight some major fixes. There are many more which are all documented in the *patches* directory of ANA.

5.3.3 Progress to date

Figure 5.6 shows the current progress made with the 32-bit port. It is difficult to estimate such progress but based on roughly how many libraries compile and how many fail to compile one can express this as a percentage of confidence. It can be estimated that 60-80% of the ATLAS specific code has been ported to ARM using ANA and is still actively being worked on. This includes compiling previously uncompiled packages as well as continuous debugging. Basic test scripts such as “Athena Hello World” run successfully. More technical examples, such as a Monte Carlo simulations of $\bar{t}t$ production from gluons and quarks ($gg \to \bar{t}t$ and $qq \to \bar{t}t$) in 14 TeV proton-proton collisions using the event generator Pythia8 [54] also run successfully. Future tests will include running a full analysis on ARM, from simulating events of a certain process (including the online data collection aspects) to running an analysis on said events, all within the Athena framework.

[2]These flags are needed due to how ARM treats the signedness of char/bit-fields as pointed out in [37].

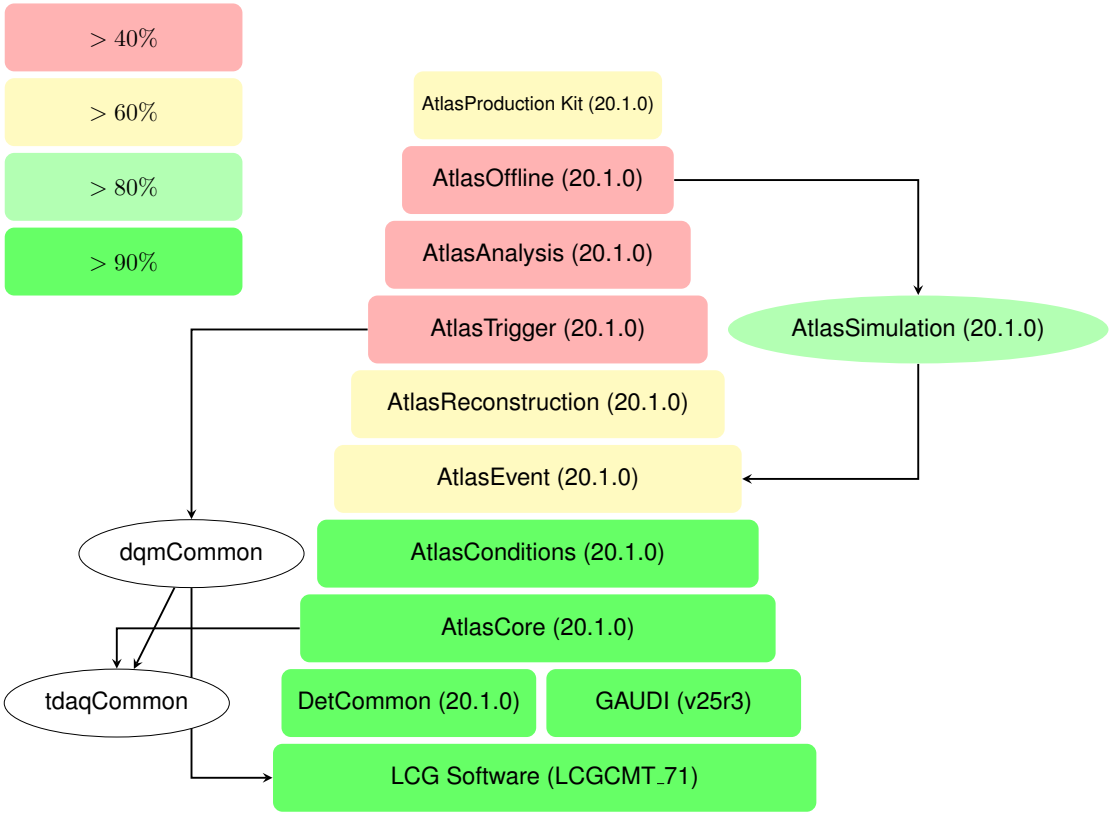


Figure 5.6: An estimate of progress made on the port of Athena to 32-bit ARM architecture.

5.4 Summary

This chapter presents the software side of HEP computing, specifically for the ATLAS experiment. There are three main steps to porting the the software to ARM. The first is to compile the external packages housed in the LCG software project. This was compiled on ARM using CMake. No significant setbacks were found (apart from Oracle DB not being compatible with ARM), although changes had to be made to the CMake steering files to satisfy the ARM compiler. The next step is to compile GAUDI. This was also done using CMake, with few changes needed to be made. Finally, ANA was presented which is a newly built framework that compiles the ATLAS specific code using CMT with minimal input from the user. The structure of the program was described as well as the nature of the patches. Using ANA, roughly 70% of the ATLAS code has been ported to ARM. Example Athena scripts making use of a variety of packages and projects run successfully for the first time on ARM.

Chapter 6

Conclusion

This thesis presents a study on using ARM processors for applications in HEP. In the past, clock speed had been one of the most important metrics in computing. While clock speed is still important, memory bandwidth has struggled to keep up with the increased processing speeds and so has also become a foremost metric. As faster supercomputers and larger server farms are built a further metric is becoming as prevalent as the previous two, that of performance per watt. This new new metric becomes crucial when designing server farms or upgrading them for the next big experiments. For the LHC this is Run 3, which is scheduled for early 2020.

Benchmark tests were performed on various ARM processors to see their current performance, and more importantly, performance per watt. It has been recognized that for ARM processors to be accepted in the ATLAS experiment, they need to be able to run Athena. Thus, this thesis presents a port of the ATLAS software stack to ARM. In doing so, a framework was written to consistently compile Athena releases on ARM.

All the benchmark results show that ARM processors are very power conservative. However, in order for people to make the decision to switch to ARM a certainty must be given. A certainty that cannot be provided from these results. According to the E/p analysis, to roughly match the performance of Hep306 enough CPU's are needed so that power savings, and therefore cost savings become negligible. Thus, it may not be worthwhile to replace traditional CPU's with these specific ARM processors.

However, there are some caveats. OS's make a large impact on performance. To compile the software stack we are limited to an ARM flavour of Fedora which clearly hasn't been optimized. Thus, benchmark results do have room for improvement. More time is needed

to fix these issues. Another consideration is that the new 64-bit architecture, providing it is as power efficient as the 32-bit architecture, should have a large effect on the viability of ARM processors in HEP computing.

The ARM architecture has matured enough that compiling HEP software poses few irrecoverable issues (such as Oracle DB). At this point it requires effort to incorporate the ARM architecture into an Athena release. Until there is a clear benefit in using ARM processors, ATLAS will not factor ARM builds into NICOS. However, they need to be assured that Athena runs on ARM. Thus, ANA has been created to provide this intermediate step, and as a proof of concept this has been successful. Athena example and test scripts have run for the first time on the ARM processor. Near future developments will see simulation, event reconstruction and a full analysis performed on the ARM architecture, within a packaged version of the Athena framework.

Chapter 7

Epilogue

For the time period following the initial submission of this thesis and the time taken to receive corrections from the examiners, major progress was made with the port of Athena to ARM. A month was spent at CERN, where the primary focus was to apply the previously gained knowledge to implementing ANA on the 64-bit architecture. ANA successfully runs on the 64-bit architecture and significant changes were made to progress the port as well as speed up compilation time. Some major improvements include the following:

- Individual packages within projects are no longer checked out from SVN using CMT. Instead, tar-balls located in lxplus are used.
- ROOT6 is now being used.
- Debugging information is no longer kept at compile time. This reduces the storage requirements by about a factor of ten and thus also speeds the process up.
- The tdaq- and dqm-common projects have now been compiled, patches have been written and tar-balls have been created.
- A testing framework that runs tests and logs results for each project has been added.

Another estimate can be made to the progress of the port in similar fashion to that done in Chapter 5. This is shown in Figure 7.1. The newer version of ANA can be found in the same place as before under the branch name “arm64”. The README files explain in detail how to use this version of the code.

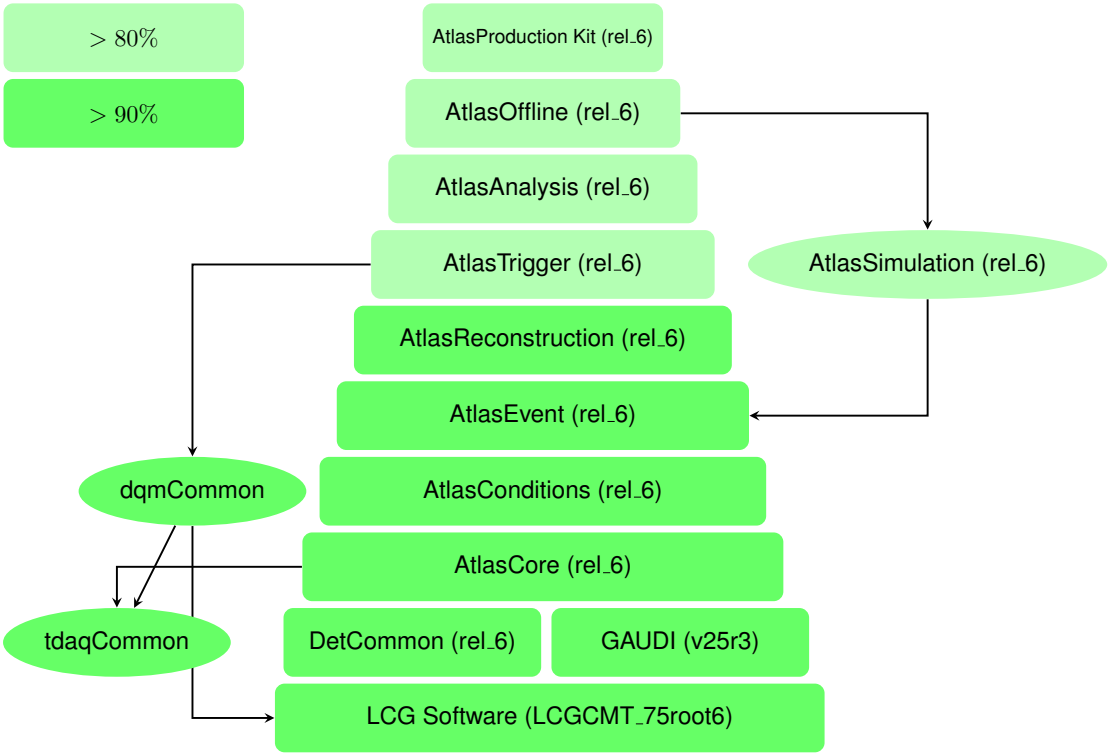


Figure 7.1: An estimate of progress made on the port of Athena to 64-bit ARM architecture following the completion and hand-in of this thesis.

Appendix A

Configuration Management Tool

ATLAS software is comprised of many different projects that are made up of packages. A package may be libraries, applications, documents etc. Many projects can form a framework. The definitions can be a bit loose. For example, ROOT is a framework in itself but in the context of an ATLAS build it becomes a package. The labeling of a project or package therefore depends on the context provided. A release is the term that is used for a completed project. In order for the many packages and projects to be maintained and periodically built, a Configuration Management Tool (CMT) is deployed to make everything more easy and consistent. This section gives an overview of CMT, which has been specifically created and maintained for HEP experiments at the LHC. For more in-depth instructions and information refer to [45].

A.1 Structure

The first thing to note with CMT is that directory structure is crucial and is therefore often the first downfall of new users. To stray even slightly from the required directory structure prevents the project from compiling. To illustrate this we have *Project A*, *package 1* and *package 2* within the project, as shown in Figure A.1. This contains the macro *without_version_directory*, which means that instead of an extra level of directories being created showing project and package versions, this information is put into a file in the *cmt* directory.

Project A contains the file *project.cmt* within the main *cmt* folder. This file is what defines

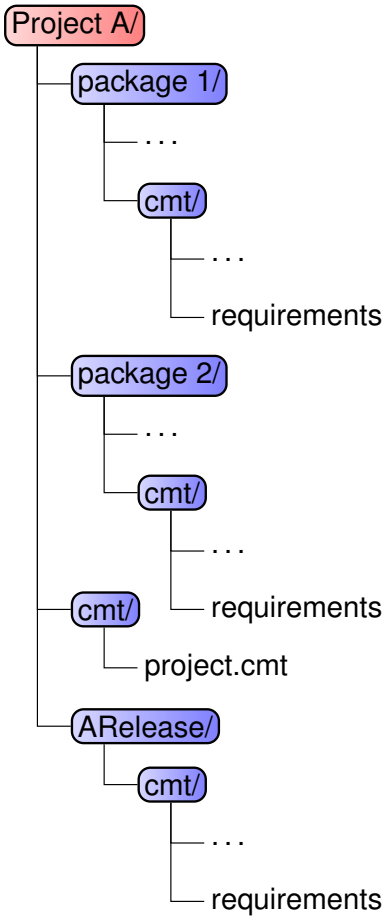


Figure A.1: The directory structure for project *A* which consists of packages *p1* and *p2*.

the name of the project being worked on, what other projects may be needed, and any overall macros that need to be applied. If a project has been compiled then it needs to be in the *CMTPROJECTPATH*, which tells CMT where to search. An example for AtlasCore is shown in Listing A.1. This needs GAUDI, DetCommon and LCGCMT_71 in its search path and will use the libraries and binaries in those projects. All the packages that will eventually be used are defined in the container package and make up a project release.

Once the *project.cmt* file has been defined, the main directory must be populated with the packages that comprise the project. These packages also have a *cmt* folder (among other folders/scripts), which contains the *requirements* file. Finally, the main directory must also contain a release package which similarly follows the CMT structure. In this *requirements* file all the packages that will be eventually be used must be defined using the “use” macro.

Listing A.1: The *project.cmt* file for AtlasCore.

```
1 project AtlasCore
2 build_strategy with_installarea
3 setup_strategy no_root no_config
4 structure_strategy without_version_directory
5 container AtlasCoreRelease AtlasCoreRelease-v*
6 use Gaudi
7 use LCGCMT LCGCMT_71
8 use DetCommon DetCommon-20.1.0
```

A.1.1 The *requirements* file

A description and explanation of the *requirements* file is needed as it forms the basis of how CMT functions; this file sets the guidelines for CMT to follow. In itself, CMT does no compiling. It is strictly a package manager. It tells the compiler where to look for other scripts, how to apply them and when to apply them. It does this in the form of macros and patterns. A pattern is a group of macros applied in certain situations, almost like a function in everyday programing. An excerpt is taken from the AtlasCxxPolicy package and shown in Listing A.1.1. Some basic examples are:

- Line 3, 6, 7. The use statement is one of the most common macros. It tells CMT where macros and patterns have already been defined and whether or not to inherit environment variables from them.
- Line 10. A macro telling CMT where to find header files.
- Line 14. If the host machine is MacOSX and default compiler is named “gcc40”, then it renames the compiler as “gcc-4.0”.
- Line 18-20. If the target system uses “gcc” and is of 32-bit or 64-bit, x86 architecture then append the flags “-m32” to the pattern “shlibbuilder”, which is defined in a different *requirements* file.

These are some basic macros. There are many and often one has to resort to the manual or look at examples should changes have to be made.

Listing A.2: An excerpt from the *requirements* file for AtlasCxxPolicy.

```
package AtlasCxxPolicy

use GaudiPolicy v* -no_auto_imports

# Setup tags etc. declared by ExternalPolicy
use ExternalPolicy      ExternalPolicy-*  External
use AtlasCompilers      AtlasCompilers-*  External

build_strategy no_prototypes
include_dirs $(AtlasCxxPolicy_root)
#-------------------------------------------------------------
# Override C++ compiler version on MacOSX 10.6 (Snow Leopard) when using gcc40
#-------------------------------------------------------------
macro cpp_name "$(cpp_name)" host-mac106&gcc40 "gcc-4.0"
#-------------------------------------------------------------
# Add support for building 32-bit libraries on a 64-bit system
#-------------------------------------------------------------
macro_append shlibbuilder                       "" \
             target-gcc&target-i686&host-x86_64 " -m32" \
             target-gcc&target-i386&host-x86_64 " -m32"
...
```

A.2 Implementing CMT

Once the above steps have been carried out for a desired project one has to implement a few basic commands to compile everything. From the container package's *cmt* directory: Create the setup and Makefiles:

```
>> cmt config
```

Tell CMT where the other packages are:

```
>> source setup.sh
```

Tell the other packages in the project to create setup and Makefiles:

```
>> cmt broadcast cmt config
```

Make all the packages using n number of cores:

```
>> cmt broadcast make -jn
```

Bibliography

[1] L. Evans and P. Bryant, *LHC Machine*, JINST 3 (2008) S08001.

[2] CMS Collaboration, S. Chatrchyan *et. al.*, *Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC*, Phys.Lett. B716 (2012) 30–61.

[3] ATLAS Collaboration, G. Aad *et. al.*, *Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC*, Phys.Lett. B716 (2012) 1–29.

[4] CMS Collaboration, S. Chatrchyan *et. al.*, *The CMS experiment at the CERN LHC*, JINST 3 (2008) S08004.

[5] ATLAS Collaboration, G. Aad *et. al.*, *The ATLAS Experiment at the CERN Large Hadron Collider*, JINST 3 (2008) S08003.

[6] I. Bird, K. Bos, N. Brook, D. Duellmann, C. Eck, *et. al.*, *LHC computing Grid. Technical design report*. 2005.

[7] G. E. Moore, *Cramming more components onto integrated circuits*, Electronics 38 (1965).

[8] G. E. Moore, *Lithography and the future of moore's law*, Proc. SPIE 2438, Advances in Resist Technology and Processing XII 2 (1995).

[9] S. H. Fuller and L. I. Millett, *Computing Performance: Game Over or Next Level?*, vol. 44. 2011.

[10] S. Glashow, *Partial Symmetries of Weak Interactions*, Nucl.Phys. 22 (1961) 579–588.

[11] S. Weinberg, *A Model of Leptons*, Phys.Rev.Lett. 19 (1967) 1264–1266.

[12] A. Salam, *Gauge Unification of Fundamental Forces*, Rev.Mod.Phys. 52 (1980) 525–538.

[13] "ATLAS Experiment." http://www.atlas.ch/.

[14] "WLCG - WorldWide LHC Computing Grid." http://wlcg-public.web.cern.ch/.

[15] R. Brun and F. Rademakers, *ROOT: An object oriented data analysis framework*, Nucl.Instrum.Meth. A389 (1997) 81–86.

[16] W. Lavrijsen, "A Python-ROOT Bridge." http://wlav.web.cern.ch/wlav/pyroot/.

[17] M. Ballintijn, R. Brun, F. Rademakers and G. Roland, *The proof distributed parallel analysis framework based on root*, CHEP03 Intl. Conf. (2003).

[18] A. Manafov, *Dynamic PROOF clusters with PoD: Architecture and user experience*, J.Phys.Conf.Ser. 331 (2011) 072052.

[19] ATLAS Collaboration, G. Aad *et. al.*, *Commissioning of the ATLAS Muon Spectrometer with Cosmic Rays*, Eur.Phys.J. C70 (2010) 875–916.

[20] ALICE Collaboration, K. Aamodt *et. al.*, *The ALICE experiment at the CERN LHC*, JINST 3 (2008) S08002.

[21] LHCb Collaboration, J. Alves, A. Augusto *et. al.*, *The LHCb Detector at the LHC*, JINST 3 (2008) S08005.

[22] E. Vermij, L. Fiorin, C. Hagleitner, and K. Bertels, *Exascale radio astronomy: Can we ride the technology wave?*, in Supercomputing (J. Kunkel, T. Ludwig, and H. Meuer, eds.), vol. 8488 of *Lecture Notes in Computer Science*, pp. 35–52. Springer International Publishing, 2014.

[23] "TOP500." http://www.top500.org/.

[24] L. Dagum and R. Menon, *OpenMP: an industry standard API for shared-memory programming*, Computational Science & Engineering, IEEE 5 (1998), no. 1 46–55.

[25] G. Edgar *et. al.*, *Open MPI: Goals, concept, and design of a next generation MPI implementation*, in Proceedings, 11th European PVM/MPI Users' Group Meeting, (Budapest, Hungary), pp. 97–104, September, 2004.

[26] G. M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, in Proceedings of the April 18-20, Spring Joint Computer Conference, AFIPS '67, (New York, NY, USA), pp. 483–485, ACM, 1967.

[27] J. L. Gustafson, *Reevaluating Amdahl's Law*, vol. 31. ACM, New York, NY, USA, May, 1988.

[28] X.-H. Sun and Y. Chen, *Reevaluating amdahl's law in the multicore era*, J. Parallel Distrib. Comput. 70 (2010), no. 2 183–188.

[29] "Arm Cortex-A series." http://www.arm.com.

[30] Fluke Corporation, *Users Manual: 287/289 True-rms Digital Multimeters*.

[31] "Intel® Power Gadget." https://software.intel.com/en-us/articles/intel-power-gadget-20.

[32] "RedHat Enterprise Linux." http://www.redhat.com/en/technologies/linux-platforms/enterprise-linux.

[33] "Scientific Linux CERN." http://linux.web.cern.ch/linux/scientific6/.

[34] "CentOS." https://www.centos.org/.

[35] "RedSleeve." http://www.redsleeve.org/.

[36] N. Rajovic *et. al.*, *Tibidabo1: Making the case for an ARM-based HPC system*, Future Generation Computer Systems 36 (2014), no. 0 322 – 334.

[37] D. Abdurachmanov, P. Elmer, G. Eulisse, and S. Muzaffar, *Initial explorations of ARM processors for scientific computing*, J.Phys.Conf.Ser. 523 (2014) 012009.

[38] S. V. Kartik, B. Couturier, M. Clemencic, and N. Neufeld, *Measurements of the LHCb software stack on the ARM architecture*, J.Phys.Conf.Ser. 513 (2014) 052014.

[39] M. A. Cox, R. Reed, and B. Mellado, *The development of a general purpose ARM-based processing unit for the ATLAS TileCal sROD*, Journal of Instrumentation 10 (2015), no. 01 C01007.

[40] M. A. Cox, R. Reed, and B. Mellado, *Affordable and power efficient computing for high energy physics: CPU and FFT benchmarks of ARM processors*, Proceedings of SAIP2014 (2015) 180–184.

[41] J. Dongarra, *Linpack benchmark*, Computer Science Technical Report CS - 89 – 85, University of Tennessee.

[42] J. D. McCalpin, “STREAM: Sustainable Memory Bandwidth in High Performance Computers.”

[43] T. Bingmann, “pmbw - Parallel Memory Bandwidth Benchmark / Measurement.” http://panthema.net/2013/pmbw/, 2013.

[44] ATLAS Collaboration, C. Meyer, *The ATLAS Tile Calorimeter Calibration and Performance*, EPJ Web Conf. 60 (2013) 20051.

[45] C. Arnault, “Configuration Management Tool.” http://www.cmtsite.net/, 2006.

[46] N. Bliss, *SVN/Subversion Version Control Crash-Course & Quick Reference*. CreateSpace Independent Publishing Platform, USA, 2013.

[47] “XRootD.” http://xrootd.org/.

[48] S. Ryu and G. Ganis, *The PROOF benchmark suite measuring PROOF performance*, J.Phys.Conf.Ser. 368 (2012) 012020.

[49] G. Barrand *et. al.*, *GAUDI - A software architecture and framework for building HEP data processing applications*, Comput.Phys.Commun. 140 (2001) 45–55.

[50] A. E. Undrus, *NICOS system of nightly builds for distributed development*, eConf C0303241 (2003) TUJT006.

[51] “CMake.” http://www.cmake.org/.

[52] “ANA-ATLAS Nightly on ARM.” https://github.com/jwsmithers/AtlasOfflineBuild-framework.

[53] J. Textor, J. Hardt, and S. Knüppel, *DAGitty: A Graphical Tool for Analyzing Causal Diagrams*, Epidemiology, 5(22):745 (2011).

[54] T. Sjostrand, S. Mrenna, and P. Z. Skands, *A Brief Introduction to PYTHIA 8.1*, Comput.Phys.Commun. 178 (2008) 852–867.